Simulating OPC UA Information Models

Lauri Saikko

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 19.3.2018

Supervisor

Prof. Valeriy Vyatkin

Advisors

M.Sc. (Tech.) Jouni Aro

D.Sc. (Tech.) Ilkka Seilonen

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Lauri Saikko

Title Simulating OPC UA Information Models

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Valeriy Vyatkin

Advisors M.Sc. (Tech.) Jouni Aro, D.Sc. (Tech.) Ilkka Seilonen

Date 19.3.2018 **Number of pages** 56+4 **Language** English

Abstract

OPC UA is an industrial communication specification that provides secure exchange of data between client and server applications. It introduces the concept of information modeling with an abstract meta model that binds semantics to data. An Information Model describes any kind of specific data in a structured way and uses the concepts of the meta model, thus advancing the interoperability of different systems on the information level.

This thesis introduces new features to a test server called the Simulation Server for simulating data created according to types defined in Information Models. The intent is to mimic the behaviour of data that is exposed by a real production server using the same Information Models. In the first part of the thesis, an algorithm for serializing an Information Model contained in an arbitrary server into a machine-readable XML format is designed. A serialized Information Model can be imported to the Simulation Server, after which it can be used to create specific data to the server. This thesis also introduces new features for configuring simulation signals in such way that the created data can conveniently be simulated.

The most important aspect of configuring simulation signals is that it can diversely be performed to the types of an Information Model. The motivation for this feature is that configuring simulation signals to types and simulating data according to the type configurations eliminates the need to configure every piece of data manually, which is beneficial especially when large amounts of data is created. The serialization feature is also used in the Simulation Server; the imported Information Models, the created data, and the simulation configurations are saved when the server is closed. Therefore, they can be restored during restart. The software designed in this thesis is merely a prototype that will be eligible for future development that will involve, e.g., designing more accurate simulation models.

Keywords OPC UA, Simulation Server, Information Model

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Lauri Saikko

Työn nimi OPC UA:n tietomallien simulointi

Koulutusohjelma Automaatio ja sähkötekniikka

Pääaine Säätötekniikka, robotiikka ja autonomiset järjestelmät **Pääaineen koodi** ELEC3025

Työn valvoja Prof. Valeriy Vyatkin

Työn ohjaajat DI Jouni Aro, TkT Ilkka Seilonen

Päivämäärä 19.3.2018 **Sivumäärä** 56+4 **Kieli** Englanti

Tiivistelmä

OPC UA on teollisuudessa käytetty tiedonsiirtomääritelmä, jonka avulla asiakas- ja palvelinsovellus voivat turvallisesti vaihtaa tietoa keskenään. Se mahdollistaa tiedon mallintamisen abstraktin metamallin avulla. Kyseinen metamalli liittää kaikkeen dataan tietyn semantiikan. Tietomallit kuvaavat mitä tahansa rakenteellista tietoa käyttäen hyödyksi edellä mainittua metamallia. Tämä varmistaa sen, että eri sovellukset pystyvät käsittelemään mitä tahansa tietomallien perusteella luotua dataa.

Tässä työssä tutkitaan tietomallien määrittämien tyyppien perusteella luodun datan simulointia testipalvelimella, jota kutsutaan simulaatiopalvelimeksi. Simuloinnin tarkoitus on jäljitellä oikean tuotantopalvelimen tarjoaman ja samojen tietomallien määrittämän datan käyttäytymistä. Työn ensimmäisessä osassa suunnitellaan algoritmi tietomallien sarjallistamiseksi XML-formaattiin. Sarjallistetun tietomallin voi tuoda simulaatiopalvelimelle, jossa tietomallin perusteella voi luoda dataa. Tässä työssä esitellään myös uusia ominaisuuksia simulaatiosignaalien konfigurointiin, jotta luotua dataa voi kätevästi simuloida.

Simulaatiosignaalien konfiguroinnin tärkein näkökulma on, että sitä voi monipuolisesti tehdä tietomallissa määritetyille tyypeille. Tämän ominaisuuden tarkoitus on se, että kun simulaatio on konfiguroitu tyyppeihin ja dataa simuloidaan tyyppikonfiguraatioden perusteella, ei kaikkea dataa tarvitse konfiguroida erikseen. Tämä on hyödyksi erityisesti silloin, kun palvelimelle luodaan paljon simuloitavaa dataa. Sarjallistamisominaisuutta hyödynnetään myös simulaatiopalvelimella niin, että sen tietomallit, luotu data ja simulaatiokonfiguraatiot tallennetaan kun palvelin suljetaan ja palautetaan uudelleenkäynnistyksen yhteydessä. Tässä työssä luotua simulaatiopalvelimen prototyyppiä voi jatkossa kehittää esimerkiksi suunnittelemalla tarkempia simulaatiomalleja.

Avainsanat OPC UA, simulaatiopalvelin, tietomalli

Preface

I would like to thank Prosys for this Master's thesis opportunity and Jouni Aro for coming up with the subject for the thesis and for giving me invaluable guidance and feedback throughout the process. I also want to thank Ilkka Seilonen and Valeriy Vyatkin for advice and feedback and my coworkers at Prosys for providing a good atmosphere to work in.

Otaniemi, 19.3.2018

Lauri Saikko

Contents

Abbreviations

A&E	Alarms & Events
API	Application Programming Interface
COM	Component Object Model
DA	Data Access
DCOM	Distributed Component Object Model
DOM	Document Object Model
ERP	Enterprise Resource Planning
GUID	Globally Unique Identifier
HDA	Historical Data Access
HMI	Human Machine Interface
HTML	Hypertext Markup Language
JAXB	Java Architecture for XML Binding
JAXP	Java API for XML Processing
JDK	Java Development Kit
MES	Manufacturing Execution System
OLE	Object Linking and Embedding
OPC	OLE for Process Control
OPC UA	OPC Unified Architecture
RPC	Remote Procedure Call
SAX	Simple API for XML
SCADA	Supervisory Control and Data Acquisition
SDK	Software Development Kit
SOA	Service Oriented Architecture
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XSD	XML Schema Definition

1 Introduction

1.1 Background

In the field of industrial automation, the communication between different systems is becoming increasingly important. Information systems, such as Enterprise Resource Planning (ERP) and Manufacturing Execution Systems (MES), require means to exchange data between each other in order to secure the flow of data gathered from industrial processes so that the processes can be supervised and controlled. OPC Unified Architecture (OPC UA), the successor of OPC Classic, is a communication protocol that provides means for applications to exchange data and guarantees interoperability. It is a platform-independent standard, according to which a client application and a server application communicate securely over a network. These applications may be field-level devices or systems that have a higher hierarchy level. A typical use case of OPC UA is that a server exposes data from an underlying data source, such as a sensor, and a client residing in one of the aforementioned higher-level information systems can access the data exposed by the server.

Presenting data in a feasible way has been one of the main motivations for introducing OPC UA. It has a common, object-oriented model for describing the semantics of data, which enables the concept of information modeling. Servers can expose more complex structured data by extending the abstract base model in order to meet the requirements set by the process whose data the server describes. For example, a simple variable can be used to present a simple physical quantity, such as temperature. The same variable can also be encapsulated in an object that presents a larger entity spanning a hierarchy, such as a motor.

When developing an application, it is often feasible to test it against a simulated system before deploying it to actual use, because a real system might not, for example, be available to be deployed at a certain time. This also applies to OPC UA applications. A simulated OPC UA application has been introduced as a Master's thesis at Prosys OPC Ltd [1]. In the thesis, an OPC UA server called the Simulation Server was designed. The Simulation Server has the functionality of an OPC UA server and a graphical user interface developed using JavaFX, a platform for developing applications using Java programming language. The motivation for the thesis was to create an OPC UA server against which OPC UA client applications can be tested and to determine whether JavaFX is a suitable platform for this purpose. The thesis resulted in a test server that mimics an actual production server and is a useful tool for offline client development. The data that the server contains consists of simple variables that have changing values that are simulated using certain mathematical signals, e.g., sine waves. However, there is a prominent deficiency in the result when it comes to providing a test server with complex enough data so that it could actually be regarded as a reflection of a real production server.

1.2 Scope and objectives

This thesis is written at Prosys OPC Ltd, a software company specializing in OPC and OPC UA products. The thesis examines the utilization of the concept of information modeling in OPC UA server applications. The purpose is to further develop and extend the Simulation Server in such way that it will support custom Information Models containing types and instances that present structured data. The Information Models are originally defined in other servers and exported to the Simulation Server. The main objective is to develop means for configuring simulation signals to types and simulating instances according to the signals configured to the types that are abstract presentations of the instances. It will hereby be possible to simulate a large number of instances without being compelled to configure each instance individually. The support for importing custom Information Models to the server is intended to provide means for having more meaningful data in the Simulation Server and achieving better utilization of the feature of simulating values. This thesis aims to answer the following research questions:

1. *How can the types and instances of an arbitrary OPC UA server be exported to the Simulation Server?*

2. *How can simulation signals be configured to types of custom Information Models in the Simulation Server, and how are the corresponding instances simulated?*

3. *How can custom types, instances, and their simulation configurations be saved so that they can be restored after restarting the Simulation Server?*

By finding answers to the research questions, the Simulation Server is intended to become not only a tool to test client applications but a prototype that can mimic the behavior of real servers better than the previous version. The values that an instance is assigned according to a certain simulation signal are not emphasized in this thesis, because the essential aspect is the entirety of types and instances and how simulation is configured to them. In other words, a simulation signal might not accurately describe the behavior of a piece of data exposed in a real production server. The important aspect is that there are instances whose values change over time so that the possibilities of using OPC UA Information Models in simulating the data of a real server in a test environment can be studied. Even though developing the user interface of the Simulation Server requires some JavaFX-related considerations, they have already been studied in detail in the thesis that introduced the Simulation Server. The emphasis of this thesis will thus be on the internal functionality, that is, the information modeling aspect. All the requirements related to the user interface and its usability are thus out of the scope of this thesis and will not be discussed.

1.3 Research methods

The main research methods of this thesis are literature review and prototyping. One of the main sources of literature is a book by Mahnke, Leitner, and Damm [2]. Also, the 13-part OPC UA specification is studied in order to ensure that the features

designed in this thesis are compliant with the specification. The literature related to Information Models mostly focuses on creating Information Models that are specific to a certain application domain. A few examples of such studies will be given in this thesis. When it comes to combining OPC technology and simulation, the related literature (e.g., [3, 4]) mostly addresses exposing data in an OPC Classic server so that the data is provided by a separate simulation software that supports advanced simulation models. OPC UA has also been used in communication between multiple simulation systems that form a simulation model when combined [5]. However, there is no literature about servers, in which simulation features are integrated to the server itself so that one could flexibly simulate data by manipulating custom Information Models in the server. Consequently, this thesis is conducted as an exploratory research examining the possibilities of the aforementioned topics in order to result in a prototype of the Simulation Server that supports simulating data according to custom Information Models.

1.4 Structure of the work

This thesis is divided into seven chapters. Chapter 2 gives an overview of the aspects of OPC UA that are relevant in the scope of this thesis. Chapter 3 explains how the feature of serializing the types and instances of an arbitrary OPC UA server into a machine-readable format is implemented. Chapter 4 proposes a strategy for configuring simulation signals to types defined in an Information Model and simulating instances created from these types according to the configurations. Chapter 5 introduces how the concepts presented in the previous chapter are used in the Simulation Server. Chapter 6 contains conclusions and discussion of future work.

2 OPC UA

2.1 Overview

OPC (OLE (Object Linking and Embedding) for Process Control) is the interoperability standard for secure and reliable data exchange between devices and systems, and it is developed and maintained by the OPC Foundation. The first release of the standard, OPC Classic, is a Microsoft Windows based technology that provides communication between software components using COM (Component Object Model) and DCOM (Distributed Component Object Model) technologies [6]. Its purpose is to provide a standardized interface to be used in communication between supervisory-level automation systems, such as HMI (Human Machine Interface) and SCADA (Supervisory Control and Data Acquisition), and field-level devices from different vendors. It introduces three non-connected specifications for accessing data, alarms and events, and historical data: Data Access (DA), Alarms & Events (A&E), and Historical Data Access (HDA).

Even though being a success, OPC Classic has several drawbacks. For instance, its dependency on Windows platform is a certain limitation because of the wide demand for a platform-independent specification. Additionally, the requirement set by many companies to expose complex data and systems is beyond the extremely restricted information modeling capabilities of OPC Classic [2]. The emergence of Service Oriented Architectures (SOA) in manufacturing systems creates challenges also in security [7].

OPC Foundation addressed these issues by introducing a new specification called OPC UA. It is a Service Oriented Architecture that has all the functionality of OPC Classic integrated into one extensible framework [8]. OPC UA is an answer to the demand for a platform-independent specification, and it also provides enhanced security. Further, OPC UA introduces the concept of information modeling with a common, object-oriented model used for describing data. Information modeling allows the meaning of any data to be described with specific semantics. This enables, for example, the data provided by field-level devices to be comprehended also in systems of higher hierarchy levels, such as MES and ERP [9]. Therefore, interoperability on the information level of applications is ensured even when automation systems are integrated vertically. The availability of a simple, abstract base model that can also be scaled to more complex models using the extensible type system allows also complex systems to be described in a structured way in OPC UA.

OPC UA has two fundamental components: transport mechanisms and information modeling [2]. As this thesis focuses only on the server side aspects of OPC UA, the transport mechanisms needed for the communication between clients and servers are not discussed any further apart from encoding of data in XML format. It is studied in chapter 3 in the context of serializing OPC UA data into XML format. Information modeling, however, is the base concept of this thesis. Thus, it is thoroughly discussed in this chapter by introducing the OPC UA meta model, the Address Space Model, and then by introducing Information Models that are used for exposing more complex data by extending the aforementioned abstract base model.

2.2 Application architecture

An OPC UA application consists of three software layers shown in Figure 1. Stack is a low-level API (Application Programming Interface), and its C/C++, .NET, and Java implementations are maintained by the OPC Foundation, which promotes the interoperability of different applications. Stack establishes the communication channel between a client and a server and implements the OPC UA services, which are discussed in the following section. The communication channel consists of three layers: a message encoding layer that defines the data serialization format (binary or XML), a message security layer that defines how the messages are secured, and a message transport layer that defines the network protocol used in exchanging messages (UA TCP or HTTP) [2].

SDK (Software Development Kit) is a high-level API that is built on top of a stack. Developing applications on top of a stack directly is difficult, because the stack code is typically hard to understand, and using it requires somewhat deep knowledge of the OPC UA specification. SDK provides new abstraction layers of the stack for application development on both client and server sides by hiding and simplifying the complex low-level functionality implemented by the stack. As a result, the application developer is not required to have knowledge of security handling, network protocols, secure channel establishment, and such topics. Instead, the developer is exposed to a simpler API with less details, using which they create their own application-specific functionality in the application layer, which is the top level of the OPC UA software layers. SDKs also often provide sample applications that ease the start of developing applications for developers that are new to the SDK in question. The SDK used in developing the OPC UA functionality of the Simulation Server is Prosys OPC UA Java SDK [10], which will be referred to as SDK from now on. It is built on top of the Java stack [11] provided by the OPC Foundation.

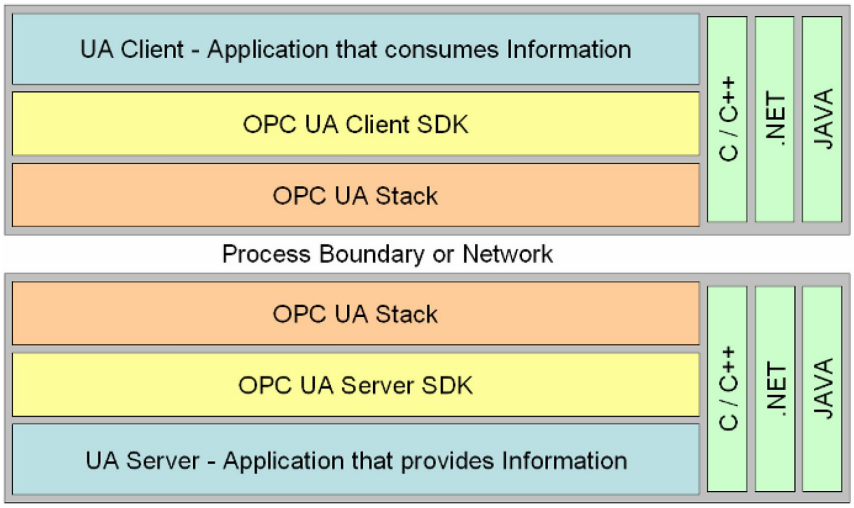

Figure 1: OPC UA application architecture. [2]

2.3 Services

Because OPC UA is a Service Oriented Architecture, servers expose their functionality using services, which are defined in part 4 of the specification [12]. The service pattern uses requests and responses, using which client and server applications communicate with each other. The client creates a request and sends it to the server that handles it, creates a response, and sends it to the client. In other words, the services are RPCs (Remote Procedure Calls) that are implemented by servers and called by clients. All of the services are not required to be provided by all servers. For example, on an embedded device with limited memory, it might not be feasible to implement every service. Using Profiles [13], it is possible to specify which services are supported.

All services have common parameters (e.g., an identifier) and service-specific ones. All responses have a service result indicating whether the request was successful or not. Services are categorized into service sets that are defined in Table 1. The services that consist a service set perform similar kinds of tasks. For example, the SecureChannel service set is related to opening a communication channel between a client and a server. In order to advance the interoperability of different OPC UA applications, a service set is defined as an abstract communication interface between clients and servers. In other words, services are independent of the underlying technologies used for the communication channel, which is why technologies can be deprecated and new ones can be adapted. The technologies are defined in part 6 of the specification [14].

Table 1: OPC UA service sets. [12]

Service set	**Use case**
Discovery	Discovering servers and reading their security configurations.
SecureChannel	Establishing a communication channel.
Session	Managing sessions.
NodeManagement	Adding, modifying, and deleting Nodes.
View	Browsing subsets of Address Space.
Query	Querying the server.
Attribute	Reading and writing Attributes of Nodes.
Method	Calling Methods.
MonitoredItem	Managing MonitoredItems used in Subscriptions.
Subscription	Managing Subscriptions.

2.4 Address Space Model

The Address Space Model is the meta model of OPC UA, and it is defined in part 3 of the specification [15]. The concepts of the model that are essential in the context of this thesis are introduced in this section. The base concept of the meta model is a

Node that presents a piece of data available on a server. Using the Address Space Model, the server exposes its consistent Address Space (the collection of data) to a client as a network of interconnected Nodes. The Nodes are described by Attributes and connected to other nodes with References. The information about a Reference is included in its source Node and its target Node. The direction of a Reference is forward from the point of view of the source Node and inverse from the point of view of the target Node (assuming the Reference is asymmetric). The References are categorized into hierarchical ones that form hierarchies of Nodes and also allow having loops and into non-hierarchical ones that do not form hierarchies. Figure 2 illustrates Nodes and their relationships to other Nodes with References.

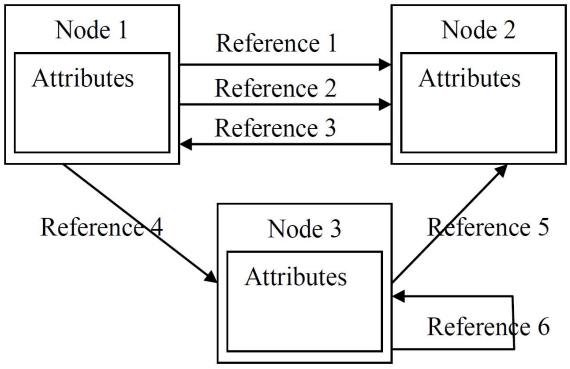

Figure 2: Nodes and References connecting the Nodes. [2]

A Node belongs to one of eight NodeClasses [15]. A NodeClass acts as the meta data for the Address Space by specifying the purpose of the Node. Each NodeClass presents either a type or an instance. An instance is created from a type that defines the characteristics of the instance, and the type is called the TypeDefinition of the instance. This relationship is established with a non-hierarchical Reference of the ReferenceType HasTypeDefinition from the instance Node to the type Node. A type is also allowed to be abstract, which means that no instance may be created from the type. Conversely, only its non-abstract subtypes may be instantiated. The NodeClasses are presented below.

Object structures the Address Space. It is mostly used to describe a real entity, such as a motor.

ObjectType is an abstraction of an Object instance. For example, an ObjectType could describe a certain type of motor. A complex ObjectType has a structure of Nodes (Variables, Objects or Methods) as its children, and they exist also in instances of the type. A simple ObjectType only defines semantics for an Object instance.

Variable contains a value related to an Object. For example, a motor Object could have temperature as a Variable. There are two kinds of Variables: DataVariables and Properties, the difference between them being that DataVariables can have child Nodes (i.e., hierarchical References to other Nodes in forward direction), whereas Properties cannot.

VariableType defines a Variable the same way as an ObjectType defines an Object. However, all the child Nodes of a complex VariableType are Variables.

ReferenceType defines the semantics of a Reference that connects two Nodes. For example, a motor Object can have a Reference to a temperature Variable, indicating that the two Nodes are related to each other with the semantics that the temperature Variable indicates the temperature of the motor Object.

DataType defines the type of the value of a Node that is of the NodeClass Variable or VariableType.

Method is a callable Node that performs an operation related to an Object that it is a child of and returns a result. For example, a motor Object could have a start Method.

View is a subset of the Address Space that restricts the number of visible Nodes and References. It is used to organize the Address Space in such way that it is more suitable to a specific use case.

OPC UA has a standard hierarchical Address Space structure that every server shall follow to promote interoperability of clients and servers. The structure is defined in part 5 of the specification [16] and shown in Figure 3. The graphical notation used to describe Nodes and References in the aforementioned figure is defined in part 3 of the specification. All the hierarchical References in the figure are of the ReferenceType Organizes, and all the Objects are of the ObjectType FolderType, except Server that is of the ObjectType ServerType. According to the standard structure, all the types are located in their respective Folders under the Types Folder depending on their NodeClass. Instances, on the other hand, are located below the Objects Folder. The Objects Folder also includes a Server Object that contains, among other things, diagnostic information of the server. Views Folder contains all the Views. Lastly, the Folders Objects, Types and Views are child Nodes of the Root Folder, which is the root Node of a server.

2.4.1 Attributes

A Node has a set of Attributes [15]. An Attribute is defined by an identifier, a name, a description, a DataType, and a mandatory/optional indicator, and it has a value according to the DataType. The set of Attributes is different for each NodeClass. Some Attributes are common to each NodeClass and some are specific to certain NodeClasses. The complete sets of Attributes for each NodeClass are defined in part 3 of the specification.

One of the Attributes common to each NodeClass is NodeId, which is perhaps the most important Attribute. It uniquely identifies a Node from other Nodes, and clients use it for referring to Nodes in service calls. A NodeId consists of a namespace index and an identifier. The namespace index is mapped to a namespace URI (Uniform Resource Identifier) in the NamespaceArray Variable that is located in the Server Object. As a small integer presenting an index is significantly shorter

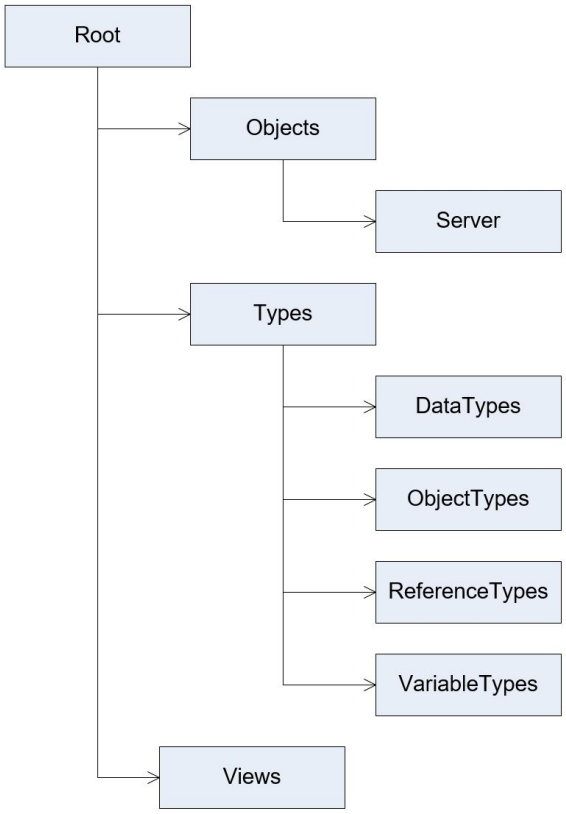

Figure 3: The standard structure of an OPC UA Address Space.

than a URI string, using indexes rather than URIs in NodeIds reduces overhead [2]. The namespace URI identifies the naming authority that assigns the identifier part and thus ensures uniqueness of NodeIds between different naming authorities. The namespace URI of the OPC UA namespace is http://opcfoundation.org/UA/, and its index is 0. The Nodes that consist the base types and the standard structure common to all servers belong to this namespace. The identifier part of a NodeId is either a number, a string, a GUID (Globally Unique Identifier), or an opaque, and it must always be unique within its namespace.

Another important Attribute in the context of this thesis is value. All Variables have a value, and VariableTypes may also have a value to indicate that it is the default value of instances that are instantiated from the type. The DataType Attribute defines the DataType of the value Attribute. The ValueRank Attribute defines whether the value is a scalar or an array (and the array dimensions in the latter case). The ArrayDimensions Attribute defines the maximum length of each array dimension if the value is an array. OPC UA has a set of built-in DataTypes whose encoding is defined by the specification; therefore, no additional information about them is required to be exposed in the Address Space [2]. There are also some simple DataTypes that are subtypes of the built-in DataTypes and encoded like their built-in supertypes [15]. The DataType of a Variable or a VariableType can also be one of the abstract DataTypes, in which case the value of the Node is encoded as one of

the non-abstract subtypes of the DataType. The built-in and some of the simple DataTypes are shown in Figure 4.

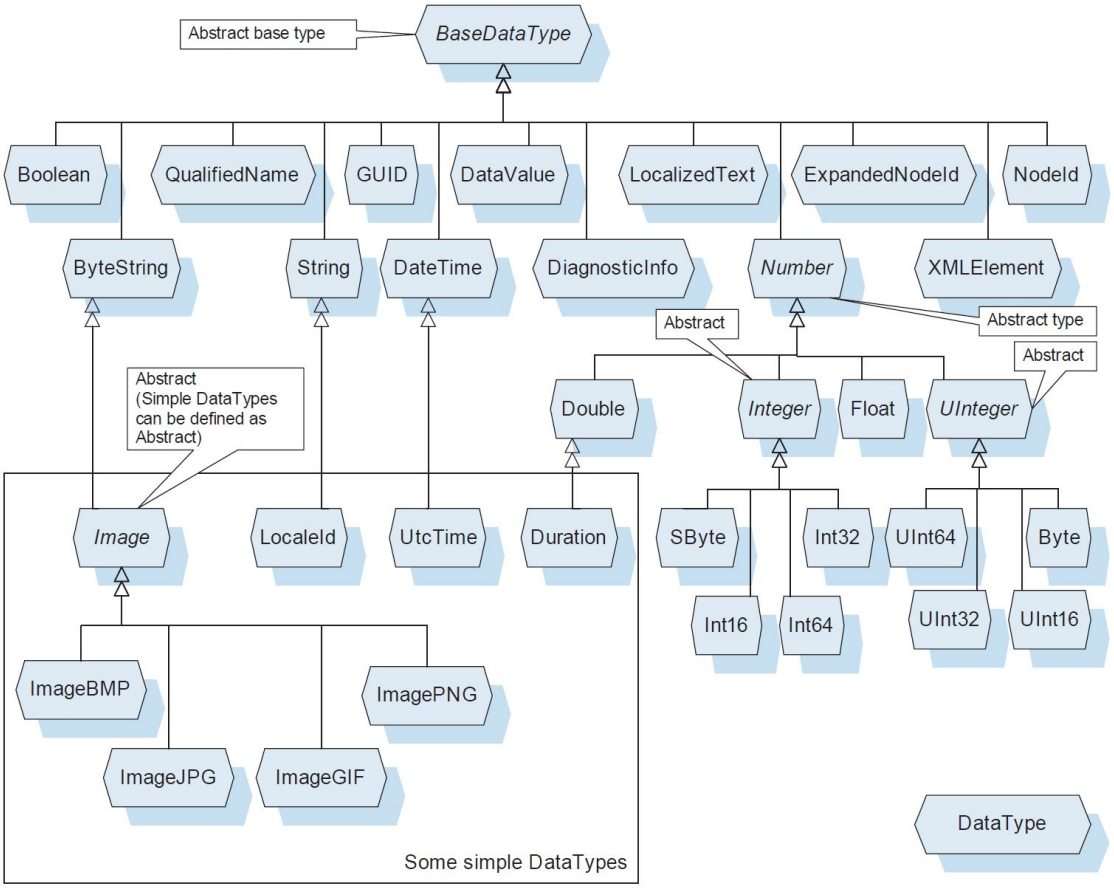

Figure 4: OPC UA DataType hierarchy. [2]

In addition to the built-in and simple DataTypes, OPC UA defines enumeration and structured DataTypes. Enumeration DataTypes are subtypes of the abstract DataType Enumeration. A value that is of an enumeration DataType is encoded as an Int32 value. However, each Int32 value has a corresponding named counterpart value that is not known from the value itself. Thus, the mapping information from the value to the named value is included in a Node that is a Property of the corresponding enumeration DataType Node. Structured DataTypes, on the other hand, are the most powerful and also the most complicated DataTypes in OPC UA. The values of DataTypes that are subtypes of the abstract DataType Structure consist of multiple fields that have a name and a value. The DataType of the value of a field may be one of the following: a built-in or a simple DataType, an enumeration DataType, or a structured DataType. Lastly, the field may be an array of any of the aforementioned types. Encoding of structured DataTypes is not defined by the specification [2]. Hence, in order to support such types, a server must define in its Address Space how they are encoded so that clients can handle them.

2.4.2 Complex types

Complex ObjectTypes and VariableTypes have a structure of Nodes beneath them, and counterparts of these Nodes exist also in every corresponding instance. The motivation for supporting complex types in OPC UA is that clients can use type information in programming their application [2]. For instance, a graphical element to a client can be programmed according to a type, after which the same element can be reused always when creating instances of the type. Another benefit of complex types is that once they are defined, new instances of the types can conveniently be added to a server by a client using the AddNodes service of the NodeManagement service set. The concept of having structured types resembles classes (i.e., templates for creating objects) that have variables in object-oriented programming languages, such as Java. This makes handling of an OPC UA Address Space more convenient in such languages. When a type is instantiated, the values of the Attributes of the instance may be restricted from the ones of the type but not vice versa. For example, a VariableType with DataType String may have a corresponding instance with DataType LocaleId (a subtype of String), but the DataType of the instance cannot be String if the DataType of the VariableType is LocaleId.

The child Nodes of complex types are instances, i.e., their NodeClass is Object, Method (the previously mentioned two in ObjectTypes only), or Variable [2]. These Nodes are not instances that present any real entity, nor contain they a real value. These kind of instances are called InstanceDeclarations, which are used to define complex types. They are referenced from a complex type Node or from another InstanceDeclaration by a hierarchical Reference in forward direction. An InstanceDeclaration is identified within a complex type by its BrowsePath. A BrowsePath is comprised of the BrowseNames of the Nodes that are on the path of Nodes from the type Node to the InstanceDeclaration. BrowseName is an Attribute that consists of a namespace index and a name, and it is unique within a type. All instances that are counterparts of an InstanceDecalaration of a complex type have the same BrowseName as the InstanceDeclaration and can thus be identified with respect to the type. NodeIds cannot be used for the same purpose, because an InstanceDeclaration Node is typically different than its counterpart Node in an instance, so the two Nodes have different NodeIds.

Clients can retrieve the counterpart Node of an InstanceDeclaration from an instance by calling TranslateBrowsePathsToNodeIds service of the View service set [12]. It takes the BrowsePath of the InstanceDeclaration and the NodeId of the instance Node that corresponds to the complex type as inputs and returns the NodeId of the counterpart of the InstanceDeclaration. It is noteworthy that BrowseNames are unique only when it comes to types. Instances of complex types may have multiple child Nodes that correspond to the same InstanceDeclaration and thus have the same BrowseName. In this case, the TranslateBrowsePathsToNodeIds service returns the NodeIds of all the Nodes that correspond to the same InstanceDeclaration.

All InstanceDeclarations have a ModellingRule, which is what differentiates them from other instances [15]. A ModellingRule specifies the use of the InstanceDeclaration in instances. There are three fundamental ModellingRules. First, Mandatory

ModellingRule means that a counterpart for the InstanceDeclaration with the same BrowsePath must exist in all instances of a complex type. Second, Optional ModellingRule implies that the instances may have a counterpart for the InstanceDeclaration but are not required to have one. Third, Constraint ModellingRule makes the InstanceDeclaration a constraint for the instances. The different types of constraints are defined in part 3 of the specification, but they are not relevant in this thesis. In an Address Space, a ModellingRule is described with an Object of the ObjectType ModellingRuleType that has a Property NamingRule containing the ModellingRule as a Mandatory ModellingRule. The ModellingRule Object is referenced by the InstanceDeclaration using a Reference of the ReferenceType HasModellingRule. ModellingRules are also extensible, meaning that vendor-specific ModellingRules can be defined.

An instance of a complex type can be used as an InstanceDeclaration in another complex type. In this case, the InstanceDeclarations of the first type have counterparts in the second type as if the first-mentioned InstanceDeclaration was a real instance. The ModellingRule of the InstanceDeclarations may be changed by tightening the original ModellingRule but not by loosening it. In other words, an Optional ModellingRule may become Mandatory, but a Mandatory ModellingRule may not become Optional. If a complex type has an Optional InstanceDeclaration and Mandatory InstanceDeclarations as its children, the Mandatory ones are not required to have counterparts in an instance if the Optional one does not. On the other hand, if the Optional InstanceDeclaration has a counterpart in the instance, all the Mandatory ones must also have. In addition to the ModellingRule, the Attributes of an InstanceDeclaration may be restricted when its complex type is instantiated.

2.4.3 Subtyping

The types of OPC UA can have subtypes. The idea of subtyping is to define a more specific presentation of a type. The primary reason for performing this is to restrict the Attributes of a supertype. Another reason is to have a type with more specifically defined semantics. In general, a subtype must always fulfill the characteristics of its supertype.

All the InstanceDeclarations of a complex type are also valid in its subtypes. Therefore, OPC UA has chosen not to copy the InstanceDeclarations to the subtype unless they are overridden, i.e., their characteristics are changed [2]. This policy has the benefit of minimizing the amount of Nodes in an Address Space. Furthermore, this is advantageous in several object-oriented programming languages, in which the variables of a superclass are automatically inherited by its subclasses, making it possible to directly reflect the structure of a subtype in a corresponding class. Overriding an InstanceDeclaration is performed by adding another InstanceDeclaration with a BrowsePath equal to the one of the overridden InstanceDeclaration to a subtype. When an InstanceDeclaration is overridden, its ModellingRule can only be changed by restricting it in a subtype. The same restrictive policy applies to the Attributes and the TypeDefinition of the InstanceDeclaration. If an InstanceDeclaration with an equal BrowsePath does not exist in any of the supertypes of a type, the

InstanceDeclaration is valid only for the type and its subtypes.

A fully-inherited InstanceDeclarationHierarchy is the combination of all the InstanceDeclarations of a complex type and its supertypes. It must be formed when the type is instantiated in order to know the full structure of the type, because the created instances must also conform to the structure. If an InstanceDeclaration is overridden in one or more subtypes, the InstanceDeclaration that is on the lowest level in the type hierarchy applies for the fully-inherited InstanceDeclarationHierarchy. The strategy for forming the fully-inherited InstanceDeclarationHierarchy is introduced in part 3 of the specification.

2.5 Information Models

OPC UA Information Models are presentations of data that specify how data of different domains is presented in an OPC UA Address Space. Although Information Models are defined only in servers, clients are able to interpret Information Models, because they use the concepts of the Address Space Model. OPC UA has a base Information Model defined in part 5 of the specification. It specifies the standard base types and instances that are common to all servers. Some parts of it are included in the Address Space Model and some are additional information, such as the standard structure of a server [2]. The base Information Model is extended by domain-specific Information Models that define types as well as constraints and instances. The types thus present data that is specific to a certain domain, and they inherit from the base types. Vendor-specific Information Models define vendor-specific types and extend the domain-specific ones. The vendor-specific Information Models can be extended further to server-specific ones. Instances of the server-specific types are used by a server to provide server-specific data. The layered Information Model architecture is shown in Figure 5. As can be seen from the figure, Information Models are built on top of the OPC UA Basis, making them independent of the underlying mechanisms used in transportation of data.

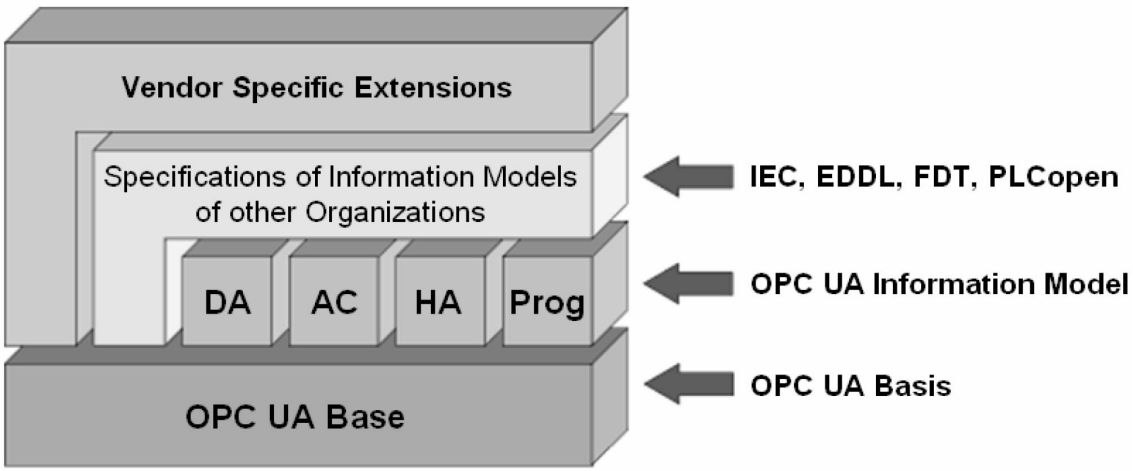

Figure 5: Layered architecture of OPC UA Information Models. [2]

The OPC UA specification includes a few domain-specific Information Models. They are intended to extend the interoperability of different applications by unifying the presentation of data in their respective domains. A few of them are introduced next as examples; however, this section does not explain their contents in detail, because Information Models are addressed in this thesis in general, and their specific characteristics does not affect the design of the new features. The Devices Information Model (DI) [17] provides presentation of automation devices in OPC UA by using well-defined types to describe the devices. The Analyzer Devices Information Model (ADI) [18] extends the Devices Information Model and defines presentation for devices in the analytical domain. The PLCopen Information Model [19] is likewise built on top of the Devices Information Model, and it defines types for the architectural models of the IEC 61131-3 standard. Several studies show that the powerful information modeling capabilities of OPC UA provide means to describe data of numerous different domains in OPC UA. For instance, the model of the building automation protocol BACnet can conveniently be mapped to OPC UA [20]. Another study demonstrates how the data models used in Smart Grids, namely the Common Information Model and the IEC 61850, can be mapped to OPC UA in order to adapt OPC UA to the Smart Grid domain [21].

3 XML serialization of an OPC UA Address Space

3.1 Introduction

In order to export the Address Space of a server to another, its Nodes must be serialized into a machine-readable format and then deserialized in the target server. One such format is provided by Extensible Markup Language (XML). This chapter introduces a design for saving the data contained in an OPC UA Address Space to an XML file in the UANodeSet format. The format is defined in part 6 of the specification. Currently, the SDK does not support serializing an Address Space; therefore, this feature is required to enable exporting the Address Space of any OPC UA server to the Simulation Server. Another use of the feature in the scope of this thesis is to save the user-defined Nodes of the Simulation Server into an XML file so that the Nodes can be automatically recreated when the server is restarted. This will be further discussed in chapter 5.

Creating Nodes to a server by deserializing an XML document that contains information about the Nodes has been studied earlier in a Master's thesis [22]. In the thesis, the format of the XML files whose processing was studied is called UA Model Design. However, deserializing an XML document that is written in the UANodeSet format is at present featured in the SDK. The OPC Foundation has also released the base Information Model in this format [23]. Therefore, the aforementioned feature is used in creating the common Nodes to the Address Space in several server applications that are developed using the SDK. As such, the same feature is also used in this thesis so that the designed Address Space serialization functionality writes Nodes in the same format. Deserializing XML data into Nodes in the Simulation Server is thus taken for granted in this thesis, and only the serialization aspect is covered.

The Address Space serialization feature of the SDK is first and foremost intended to be used with servers that are developed with the SDK, because the feature will become a part of it. It will also eventually be integrated to the Prosys OPC UA Client [24], after which it is possible to save the Address Space of a server that is developed using another SDK by creating a session to the server using the client application and then by using the integrated feature.

3.2 Extensible Markup Language (XML)

XML is a simple and flexible text format used for storing and transporting data [25]. It only specifies how to describe data, unlike Hypertext Markup Language (HTML), which is about how to display data. An XML document consists of elements that have an opening tag and a closing tag, attributes that define the elements, and content for the actual data. The following is a valid example of an XML element: <element attribute="value">content</element>. Presenting structured data is possible by nesting elements, that is, an element may have child elements, but they may not overlap. In other words, each closing tag must match the most recent opening tag that has no matching closing tag. The above mentioned facts result in XML being

readable and understandable also for humans. Additionally, XML is extensible, which means that it is allowed to define domain-specific tags that suit describing particular data. The extendability aspect makes XML a sensible language also for storing data specific to the OPC UA protocol.

An XML document may have an XML prolog written in the following format: <?xml version="1.0" encoding="UTF-8"?>. This element defines the version and the encoding used in the document. After the optional XML prolog, the root element is defined. It is the top-level element that every other element is a child element of. For all of the elements in a document, it is possible to define a namespace. The idea of this is to associate an element with a namespace URI, in the context of which the name of the element is unique. This allows differentiating elements from ones that have an equal name when merging XML documents. The default namespace for elements in a document may be defined in the root element with an attribute "xmlns". The default namespace can be overridden by specifically defining a namespace for other elements.

3.3 XML Schema Definition (XSD)

The structure of an XML document is defined by an XML schema. The language that an XML schema uses is called XML Schema Definition (XSD), which is written in XML. A schema is an abstract presentation of the elements that can be included in an XML document. It specifies, for example, the attributes and child elements of each element of a certain name. A "targetNamespace" attribute, the value of which defines the target namespace of the schema (the namespace that the elements defined in the schema belong to in a corresponding XML document), can be placed in a "schema" element, which is the root element of each schema. The abstract data model of XML Schema Definition is defined in [26]. The XML Schema Definition also defines a set of data types [27] that restrict the values of attributes and text content of elements. The support for data types is particularly useful in the OPC UA context, because a number of the data types can be directly mapped to OPC UA DataTypes. This enables values of these OPC UA DataTypes to be able to be written as defined by the constraints of the XSD data types.

The elements used in an XML schema are of the namespace http://www.w3.org/2001/XMLSchema. In a schema, an "element" element that has a "name" attribute is used to indicate that an element of the name specified by the "name" attribute may exist in a corresponding XML document. The same element may also have attributes defining constraints for the element. For instance, a "type" attribute specifies the type of the content of the element. Similarly, an "attribute" element is used to define an attribute for an element, and it can also be constrained with attributes. The value of the "type" attribute of "element" and "attribute" elements may be one of the XSD data types or a user-defined type element. The latter can be a "simpleType" element, an element that specifies constraints for the value of an attribute or the text-only content of an element. The user-defined type can alternatively be a "complexType" element, which is the same as a "simpleType", except it also defines constraints for child elements that it may have. However, the type of an "attribute" element may

not be a complex one, because attribute values can only contain text.

3.4 XML Schema Definitions for OPC UA

The OPC Foundation provides XML schemas that define the format for serializing OPC UA Nodes. There are two schemas that are required for completely describing the Nodes included in an Address Space. These schemas are introduced in the following subsections.

3.4.1 UANodeSet.xsd

The UANodeSet.xsd schema [28] first and foremost defines how Nodes are written in the UANodeSet format. The target namespace of the schema is http://opcfo undation.org/UA/2011/03/UANodeSet.xsd. The most important element in the schema is "UANodeSet", which is used as the root element in the corresponding XML documents. It defines, for example, a "NamespaceUris" child element that lists the namespace URIs corresponding to the indexes used in NodeIds and BrowseNames. Indexes are used instead of URIs in order to minimize the size of an XML document. With an "Aliases" element, one may define aliases for NodeIds in order to make a document more readable. The "UANodeSet" element also contains information about Nodes as a sequence of child elements. An element that describes a Node is named according to the NodeClass of the Node. The type that is associated with the element is a complex type, which defines how the Node Attributes that are specific to its NodeClass are stored to attributes and child elements of the Node element.

The attributes and child elements of the Node element containing the value of a Node Attribute have the same name as the Attribute whose value they present. As explained previously, if the DataType of an Attribute has a suitable counterpart in XSD data types, the value of the Attribute can be written to the value of an attribute or to the text content of an element using the XSD data type as a constraint. For example, values of Attributes of the DataType Boolean can be written using XSD data type boolean. Otherwise, a simple or a complex type is used to define how all the information that the value of an Attribute encompasses is stored. This is the case with, for example, DataTypes NodeId and LocalizedText.

The complex type UANode is the base type used for describing Nodes of each NodeClass. It contains information about storing values of all the Attributes that are common to every NodeClass. Additionally, it defines that the information about References to other Nodes is included in a "References" element that contains a sequence of "Reference" elements that present individual References. The complex type associated with the "Reference" element specifies that it includes information about the ReferenceType of the Reference, the NodeId of the other Node associated with the Reference, and the direction of the Reference.

The types used for specific NodeClasses extend the base type UANode indirectly. The types that directly extend it are UAType and UAInstance, which are the base types that contain information about Attributes common for all types and instances, respectively. The types used for each specific NodeClass extend UAType and UAIn-

stance. UADataType, UAObjectType, UAReferenceType, and UAVariableType are the types corresponding to DataType, ObjectType, ReferenceType, and VariableType NodeClasses. They extend UAType. UAMethod, UAObject, UAVariable, and UAView, on the other hand, define how to write of Nodes of the NodeClasses Method, Object, Variable, and View by extending UAInstance.

In addition to defining how different Attributes of a Node are written in XML, UANodeSet.xsd also states how enumeration and structured DataTypes are written. There is additional information related to these DataTypes, which is not included in their Attributes. Therefore, this information is stored in the element presenting a DataType Node using complex types DataTypeDefinition and DataTypeField. They define an abstract presentation of the DataTypes. DataTypeDefinition includes a sequence of elements defined by DataTypeField. In case of enumeration DataTypes, DataTypeField defines the named counterpart of an Int32 value. When it comes to structured DataTypes, it specifies the name and the DataType of a field. An element defined by DataTypeDefinition is required for enumeration and structured DataTypes in order for design tools to be able to automatically create serialization code for these DataTypes [14].

3.4.2 Opc.Ua.Types.xsd

The Opc.Ua.Types.xsd schema [29] defines how types of OPC UA are written in the UANodeSet format, and its target namespace is http://opcfoundation.org/UA/2008/02/Types.xsd. The types consist of DataTypes that are used for writing the value Attribute of Variables and VariableTypes and structures, such as requests and responses, used in different services. Only the former are used in the scope of this thesis, as they are related to serializing Nodes. As explained in the previous section, the values of Node Attributes are written to elements or attributes, which are defined based on the DataType of the Attribute. However, the value Attribute is a special kind of Attribute in the sense that it does not have a pre-defined DataType. Therefore, the element "Value" containing the value of a Node in the previously introduced UAVariable and UAVariableType complex types is defined so that it may have any kind of content. This enables values of any of the numerous OPC UA DataTypes to be written to the "Value" element.

The name of an element that presents a value as a child element of the "Value" element is the name of the DataType of the value. Therefore, Opc.Ua.Types.xsd defines an element for each OPC UA DataType so that the name of the element is the name of the DataType. The type of such element is either an XSD data type or a simple or a complex type, similarly as in the case of writing values of other Attributes. The simple and complex types that are used for these elements are also defined in Opc.Ua.Types.xsd. There are thus simple and complex types used to define storing values of the same DataTypes in both schemas, the difference between them being that the other types (in Opc.Ua.Types.xsd) are used for writing values of the value Attribute, whereas the others (in UANodeSet.xsd) are used for writing values of all the other Attributes.

Although each structured DataType also has an element named after the DataType,

structured DataTypes are not written to the "Value" element as such. Instead, they are written in an "ExtensionObject" element. It is defined by a complex type that includes the NodeId of the DataType in a "TypeId" element, and the value as a child element of a "Body" element. Each structured DataType is associated with a complex type that defines that the value of each field is written in an individual child element. The type of each child element is the type that is used to define how basic values of the DataType of the field are written.

Opc.Ua.Types.xsd also defines additional elements for writing values that are one-dimensional arrays. These elements are specific to the DataType of the value. The name of each element is "ListOf" followed by the name of the DataType. Each element is defined by a complex type that specifies that the individual values are written as a sequence of child elements that have the same name and type as the element that is used for writing scalar values of the same DataType. Multi-dimensional values, on the other hand, are written in a "Matrix" element. The type associated with it is a complex one that contains two elements. The first one has a sequence of child elements that present the length of the dimensions of the array as Int32 values. The second one contains a sequence of the actual values that are flattened into a one-dimensional array starting with the highest rank dimension.

Another vital element defined in the schema is "Variant". The element presents a Variant, which is a union of all the built-in and structured DataTypes of OPC UA and of arrays of them. A Variant can also contain arrays of Variants. Therefore, it is possible to have different combinations of values of different DataTypes as a value of a Node or as a value of a field of a structured value when the individual values are wrapped in a Variant. The complex type associated with the "Variant" element specifies that the element contains a sequence of child elements, which present the content of the Variant. Although a Node value is also a Variant per se, such values are not written inside a "Variant" element.

3.5 Java Architecture for XML Binding (JAXB)

Serializing OPC UA Nodes into XML data using the Java SDK requires means for writing XML documents with the Java programming language. Fortunately, XML and Java are recognized as an ideal combination to be used for exchanging data in numerous applications. This is due to the fact that XML has become the standard for exchanging data between systems, and Java provides a suitable platform for building different applications [30]. Java Architecture for XML Binding (JAXB) [31] is a Java API that provides means for accessing data written in XML format using Java programming language. In other words, it automates the mapping between XML elements and Java objects. Therefore, using JAXB is extremely profitable, as Java developers are not required to have expertise in XML.

The convenience of JAXB can be further demonstrated by introducing a few alternative technologies for coupling Java and XML provided by the Java API for XML Processing (JAXP) [30]. The Simple API for XML (SAX) parses each piece of an XML document and exposes the content to the application, but no data is saved to memory. The Document Object Model (DOM) creates a tree of objects

from the XML data, saves it to memory, and allows the objects to be accessed and manipulated. Therefore, SAX and DOM are merely lower-level APIs for parsing XML data, whereas JAXB is a higher-level API for creating Java objects, and it supports conversion of data in both directions.

The different operations that are performed when using JAXB is called the JAXB binding process [32]. This first step of the process is generating Java classes from an XML schema using a JAXB binding compiler. The generated classes correspond to the complex types of the schema, and the variables of the classes correspond to the attributes and child elements of the types. As such, after creating objects (i.e., instances of the classes) and by setting values to their variables, one can create an XML document that is valid according to the schema. This operation is called marshalling. The inverse operation, creating Java objects from XML data, is called unmarshalling. Both marshalling and unmarshalling involve optional validation, which means verifying that the created XML data or the XML data used for creating Java objects meets the constraints of the schema. The JAXB binding process is illustrated in Figure 6. Marshalling, unmarshalling and validation also require a binding framework that provides these features. The framework is provided by JAXB annotations that specify the mechanisms for handling the aforementioned processes in the generated classes.

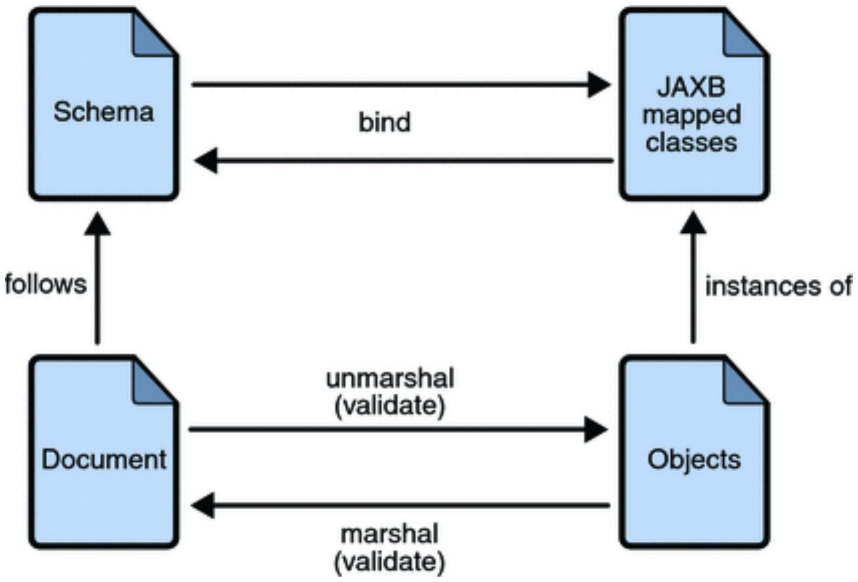

Figure 6: JAXB binding process. [32]

3.6 Generating JAXB classes

As stated in the previous section, saving data to an XML document using JAXB requires generating the Java classes from which objects are created. This can be performed using xjc [33], which is part of the Java Development Kit (JDK), so no external tools are required. xjc is a simple JAXB binding compiler that creates Java

classes with JAXB annotations from an XML schema. Using it is extremely easy, as, given that no extra options are used, the Java classes are generated from a schema with the following command on the command line: xjc <schema file>. Figure 7 describes the architecture of JAXB and how generating classes is related to it.

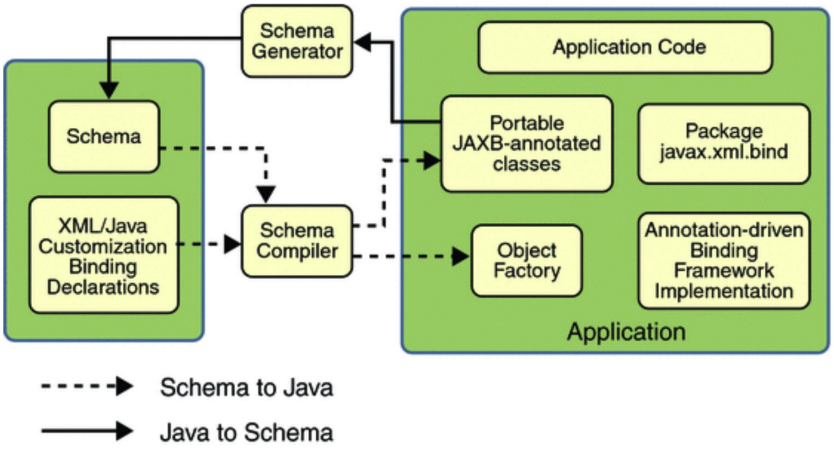

Figure 7: JAXB architectural overview. [32]

3.6.1 Data type bindings

When Java classes are created from an XML schema, the XSD data types used in the schema are bound to certain Java data types. As all the XSD data types have Java counterparts, the JAXB binding compiler does not generate classes for them, unlike for the user-defined complex types. As the text-only data defined by simple types is restricted using the XSD data types, classes for simple types are not generated either. The default XML-to-Java bindings are defined in [34].

When using the SDK, the values related to Node Attributes are presented as Java objects. In order to be able to set such object to the variable of a JAXB object, the class of the object presenting the value must match the class of the variable. In some cases, this condition is met. For example, values of DataType String are described using class java.lang.String. The same class is also the Java counterpart of XSD data type string, which is used for OPC UA DataType String in the schemas. Therefore, the default bindings of the JAXB binding compiler are sufficient.

If the class used in the SDK is not the same as the one that is bound to the XSD data type, the object provided by the SDK needs to be converted into an object of the correct type. For example, OPC UA DataTypes Byte, UInt16, UInt32, and UInt64 use classes that are provided by the OPC UA Java stack rather than the Java Class Library, so the classes bound to the XSD data types used for values of these DataTypes are different than the classes used in the SDK. In order to avoid manually converting the value obtained with the SDK into the correct class, the JAXB bindings can be customized so that the variables of the JAXB classes actually use the same classes as the SDK.

3.6.2 Customizing bindings

Customizing JAXB bindings is performed by passing a customization file to the JAXB binding compiler. There are a number of aspects that can be customized, but in this particular case, the customization file contains <javaType> binding declarations that customize the conversion of XSD data types to and from Java classes [35]. In this type of declaration, one must define an XSD data type and the Java class that it will be bound to. If this is an application-specific class, one must also specify how to handle objects of this type during marshalling and unmarshalling. This can be performed by defining print and parse methods that specify how this type is converted to and from a string in an XML document during marshalling and unmarshalling, respectively. Alternatively, one can create an implementation of the abstract class javax.xml.bind.annotation.adapters.XmlAdapter. The class contains methods marshal and unmarshal that replace the print and parse methods. The latter policy is used in this thesis.

The customization feature is used with OPC UA DataTypes Byte, UInt16, UInt32, and UInt64, the corresponding classes of which are UnsignedByte, UnsignedShort, UnsignedInteger, and UnsignedLong of the package org.opcfoundation.ua.builtintypes. The customization file for these DataTypes is presented in Appendix A, and an example adapter for UnsignedInteger is provided in Appendix B. As can be deduced from the adapter example, customizing data types is feasible when the desired Java class contains methods that make it convenient to present an object as a string and to parse a string into an object. Otherwise, it might be easier not to use customization and to convert the object into an object of the class defined by the default bindings.

3.7 Address Space serialization algorithm

This section explains the strategy used in serializing an OPC UA Address Space into an XML document. The JAXB classes referred to in this section are created from the two previously presented schemas using xjc. The marshaller that turns the objects created from the JAXB classes into XML data requires an instance of the class javax.xml.bind.JAXBContext, which provides an entry point to the JAXB API. The JAXBContext instance requires information about path to the classes or to the packages that contain the classes that it is supposed to be able to manage. Here, the path information is given to the packages that include the classes generated from the two schemas. Classes generated from each schema is located in a separate package, because the schemas result in classes that have the same name. The following subsections provide the specifics of turning the data included in an Address Space and provided by the SDK as Java objects into objects that are instances of the JAXB classes.

3.7.1 Nodes

The highest-level aspect of serializing an Address Space is browsing every Node that is to be saved and writing the data contained in the Nodes to objects of the generated classes. This requires first an object of the class UANodeSet to be created, because

the class presents the complex type, which specifies that every piece of data contained in an Address Space is stored to a root element that the aforementioned complex type defines. Therefore, the same object is eventually passed to the marshaller, and the marshaller marshals the object into XML data. Before performing this, all the data to be stored is added to the object.

Most importantly, the UANodeSet object contains information about Nodes. Adding this information is performed by browsing the whole Address Space in order to find all the Nodes that should be included. The user may choose to save every Node of a server or they can choose one or multiple namespaces whose Nodes will be saved. The resulting "UANodeSet" element will have all the namespace URIs of a server written in the child elements of the "NamespaceUris" element. In order for the correct URI to be found for any index, writing all the URIs of a server regardless of whether indexes corresponding to all of them are used is a feasible policy. It is noteworthy that the indexes corresponding to the namespace URIs in the created XML document are valid only in the server whose Address Space was saved to the document. The indexes are only used for mappings within the document, and the namespaces may have different indexes when the document is loaded to another server depending on which indexes are already reserved. Moreover, NodeIds and BrowseNames belonging to the OPC UA namespace do not require information about their namespace, that is, deserializers automatically assume that the index of a NodeId or a BrowseName without an index is 0.

In section 3.4.1, it was specified that Nodes of different NodeClasses are written in an element whose type is a complex one. Therefore, the information container of a Node in JAXB is an object that is created from a generated class. The SDK objects that depict Nodes thus need to mapped into corresponding JAXB objects. This a somewhat straightforward process, because the Attribute values that need to be set to the variables of the JAXB objects can easily be retrieved from the SDK objects. All Nodes to be serialized are found by browsing the target Node of each hierarchical Reference of each Node starting from the Root Node of the server. In order to prevent getting stuck in a loop while browsing the Address Space, information about which Nodes have already been browsed is being held during the process.

When an object describing a Node is created, information about its References is also included using objects of the classes related to them. However, information about each Reference should be written only once, meaning that the Reference is only in the References list of either the source Node or the target Node [14]. Following this guideline will minimize the size of the resulting XML document. The deserializers that read documents and build an Address Space according to the data do not need to have the information about a Reference twice, as they shall automatically add information about the Reference to both Nodes that the Reference concerns.

In this thesis, it is determined that the information about hierarchical References shall only be added to the target Node and about non-hierarchical References to the source Node. The reason for this is that the source Node of a hierarchical Reference does not need information about any of its child Nodes. In fact, if the source Node specified a hierarchical Reference in forward direction, the target Node should also be defined in the same document, which is not desired. For example, if a Node

has a component (defined by a hierarchical HasComponent Reference), it should be possible to define the component in another document without the original document having to know about it.

For non-hierarchical References, the policy is inverted. The non-hierarchical References do not form a hierarchy, so it is appropriate to include them in forward direction. On the other hand, writing them in inverse direction is typically not feasible. For example, in the case of HasTypeDefinition Reference, all the Nodes that are instances of a type would need to be included in the same document in which the type is introduced. The direction in which certain types of References are written can be changed. If the direction for a non-hierarchical ReferenceType is overridden, the References of the ReferenceType are followed in addition to hierarchical References when browsing the Address Space. The reason is that writing References of the ReferenceType in forward direction implies that the target Nodes of such References are not a target Node in any hierarchical Reference. A use case for this feature will be presented in section 5.4.1.

3.7.2 DataTypes

As presented in section 3.4.1, the abstract presentations of enumeration and structured DataTypes in the UANodeSet format are defined using the complex types DataTypeDefinition and DataTypeField. Therefore, the presentations are written using corresponding classes. In the case of enumeration DataTypes, getting the necessary data is easy, as it is included in a Property Node of the type Node. The fields of structured DataTypes, however, are not directly defined in a Node.

OPC UA specifies two data encodings that are used to encode messages exchanged between a client and a server: OPC UA Binary and OPC UA XML [14]. They define the encoding for values of different DataTypes. As structured DataTypes require additional information about their encoding to be exposed in the Address Space, each non-abstract structured DataType Node refers indirectly to DataTypeDictionary Variables that include information about the encoding of enumeration and structured DataTypes. They are used by clients in interpreting the values that they receive and also in constructing values to be sent. The encoding information is included in both OPC UA Binary and OPC UA XML formats, and each Information Model defining its own types also defines its own DataTypeDictionary Variables. In this thesis, the ones in the OPC UA XML format are used for getting the information about structured DataTypes. The value of a DataTypeDictionary presenting the OPC UA XML format is an XML schema encoded as a ByteString. In such schema, structured DataTypes are described using complex types, as explained in section 3.4.2. When the value of a DataTypeDictionary is parsed, the names and DataTypes of the fields of structured DataTypes can be written to DataTypeField objects.

3.7.3 Values

The Node values that are of a DataType whose SDK counterpart class is also used by the variables of JAXB classes are already discussed in the previous sections. Therefore, this section focuses on those values that are provided in the SDK by an

object that needs to be converted into another object. LocalizedText and NodeId are good examples of DataTypes of these kinds of values, because they have a structure internally. Values of these DataTypes are converted from SDK objects to JAXB objects similarly as in the case of Nodes, i.e., by reading different pieces of data of the SDK objects and by setting them to the values of the variables of the JAXB objects. If a value is a one-dimensional array, an object depicting a list of values is created. The elements that the list contains are written using the same class that is used for scalar values of the same DataType, as they are defined using the same type in Opc.Ua.Types.xsd. These individual values are also used in a Matrix object if the value is a multi-dimensional array.

Because values of simple DataTypes are encoded like their built-in supertypes, Opc.Ua.Types.xsd defines these values to be written using the types that also their supertypes use. Thus, they also use the same classes in JAXB. On the other hand, values of Nodes that have an abstract DataType are written using the class related to the DataType of the actual value as if the DataType of the Node was not abstract.

3.7.4 Structured values

Writing structured values is the most challenging part of serializing Nodes into XML data. This is due to the fact that the base Information Model defines dozens of different structures that have different sets of fields, and a general way to save structured values without knowing the fields needs to be found in order to avoid handling every different type of structured value individually, which would require an enormous amount of work. The key to resolving this challenge is using reflections. The reflection feature of Java is used for accessing objects, their variables, and methods at run time without knowing the names of any of these at compile time.

When a structured value is serialized, the JAXB class that is the counterpart of the complex type that defines the values of the structured DataType needs to be determined. Fortunately, the names of the SDK classes used to store structured values are equal to names of the corresponding JAXB classes. Thus, the correct JAXB class is retrieved with the name of the SDK class from the correct package using reflection. Then, the value of each field is retrieved using reflection on the SDK object. This is performed by calling a method by the name of "get" followed by the name of the field. When a field value is converted into an object suitable for the variable of the JAXB object, it can be set to the variable by calling the method "set" followed by the name of the field likewise using reflection.

Converting a field value object into one required by the variable of the JAXB object is performed somewhat similarly as if the value was the value of a Node. The field value may be of one of the built-in or simple DataTypes whose handling has already been discussed. If it is a structure, the steps mentioned in this section are repeated, as the value is written using the class that presents the complex type associated with that structured DataType. A structured field value can also be defined as an ExtensionObject, an encoded value that may contain a value of any structured DataType. Then, the value needs to be decoded before writing it using the class corresponding to the complex type that defines an "ExtensionObject" element.

To make matters more complicated, the names of the XML elements that present field values and thus the names of the corresponding variables of the JAXB classes are equal to the name of the field rather than to the name of its DataType. Therefore, an object that is specific to a field name is created to wrap the previously created object containing the field value.

Writing values of structured DataTypes defined in a custom Information Model is troublesome, because the Opc.Ua.Types.xsd schema only includes information about the structures defined in the base Information Model. The schema containing information about structures defined in a custom Information Model can be found only in the DataTypeDictionary Variable defined by the Information Model. Therefore, there are generated classes only for the structured DataTypes of the base Information Model. One solution to this problem would be to extend the serialization feature so that it allowed configuring additional JAXB classes. These classes would be generated from the aforementioned schema by the user and used to serialize custom structures to XML. However, writing values of custom structured DataTypes is determined to be out of the scope of this thesis.

4 Simulation strategy

This chapter introduces the new simulation features of the Simulation Server from the OPC UA point of view. Although the version of the Simulation Server designed in this thesis is merely a prototype, all the prospective features to be added on top of it will be based on the concepts presented in this chapter. All the simulation signals that will be mentioned are similar to the ones in the original Simulation Server, and they cannot be created from real data sources. This is because the objective of this thesis is not to design more accurate simulation models but to define semantics for configuring simulation to Nodes especially on the type side. The term TypeDefinition is from now on used to refer to the type counterpart of an instance. It may be an InstanceDeclaration rather than the type Node that the instance refers to with a HasTypeDefinition Reference.

4.1 Introduction

The foundation of the new simulation features is that a simulation signal can be configured to VariableTypes and Variable InstanceDeclarations of complex VariableTypes and ObjectTypes. When instances of such types are created, the values of the Variables in the corresponding instances can be simulated using the signals configured to TypeDefinition Nodes. The benefit of this feature can be demonstrated with an example: there is an ObjectType that describes a boiler. The boiler type has a Variable as an InstanceDeclaration. It depicts the temperature of the fluid inside the boiler. If the user creates numerous identical boiler instances to the server, the temperature Variable of each instance is automatically simulated according to the signal configured to the InstanceDeclaration. Thus, the user does not have to configure simulation to each instance separately.

There are three different practices when it comes to simulating a Variable instance. First, the simulation signal used to simulate the instance can be configured to the TypeDefinition counterpart of the instance, which is what was already mentioned. Second, the signal may be configured to a TypeDefinition Node that the TypeDefinition counterpart of the instance is a more specific presentation of. These two strategies are presented in section 4.2. The third option is to override the previously mentioned two type-based configuration patterns by configuring a simulation signal to the instance itself. This strategy is presented in section 4.5.

The simulation signals are separated from the Node objects into objects that contain the current value and the parameters that will determine the next value. The reason for separating the simulation object from the Node that the simulation is configured to is that the Node is not necessarily the one that will use the simulated values. In fact, simulated values are never set to TypeDefinition Nodes. Having the simulation separated from the Node is beneficial also because it allows testing simulation models separately from any OPC UA functionality.

4.2 Type-based simulation

As mentioned in the previous section, there are two practices for simulating instances according signals configured to TypeDefinitions. In addition to using the signal configured to the TypeDefinition counterpart, an instance can also be simulated according to the signal of another TypeDefinition if the counterpart TypeDefinition is a Variable InstanceDeclaration. In this case, the alternative TypeDefinition is either the VariableType of the Variable or another Variable of the same VariableType. In the latter case, the alternative Variable is an InstanceDeclaration in a complex type so that the original Variable is an instance counterpart of this Variable. In other words, an instance of this complex type is directly or indirectly used as an InstanceDeclaration in the complex type that the original Variable is a part of. This feature can be demonstrated using types that are defined using the graphical notation in Figure 8 as an example. The figure introduces two different VariableTypes, one of which is used as an InstanceDeclaration in the other. There is also an ObjectType that has an instance of the latter VariableType as an InstanceDeclaration. The BrowseNames of the Nodes are equal to their DisplayNames shown in the figure.

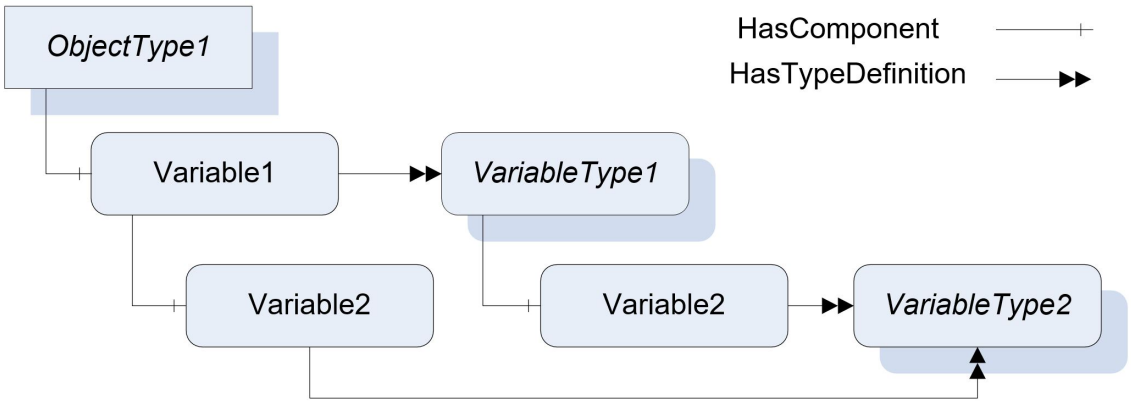

Figure 8: Example types.

If an instance of ObjectType1 is created and its counterpart for Variable2 is to be simulated, simulation can be configured to three different TypeDefinition Nodes. The first option is Variable2 InstanceDeclaration of ObjectType1. Alternatively, the signal of Variable2 of VariableType1 can be used. Lastly, the signal configured to the actual type of Variable2, VariableType2, can be used. In general, each aforementioned Node is either the type of the Node whose instance counterpart is to be simulated or a more specific presentation of the type (an InstanceDeclaration). In the latter case, the Node is a part of a larger entity, i.e., a complex type. Such entity is either the one having the original InstanceDeclaration or a smaller one that forms the first-mentioned entity directly or indirectly.

4.2.1 Use case example

The motivation for allowing the simulation of an instance to be based on signals configured to different TypeDefinitions can be exemplified with an analogy that could

be a use case for it. There is a sensor that measures temperature, and the sensor is kept at room temperature by default. The temperature measured by the sensor can be simulated using a certain formula. If the sensor is put in a container that has heated fluid in it, the formula is overridden by another one. Further, if the container that contains the sensor is put to a furnace, the measurements change, and the formula is different once again. In OPC UA, this could be described by having a VariableType (presenting the sensor, the value of which is the measurement), which becomes a part of an ObjectType (presenting the container) as an InstanceDeclaration. This ObjectType becomes an InstanceDeclaration of yet another ObjectType (presenting the furnace). Such types are visualized in Figure 9. It is obvious that although there are three different formulas used for the measurements in the different conditions, the one used is the one that defines the measurement for the furnace. In other words, the simulation that is configured to the most specific presentation of an entity is used.

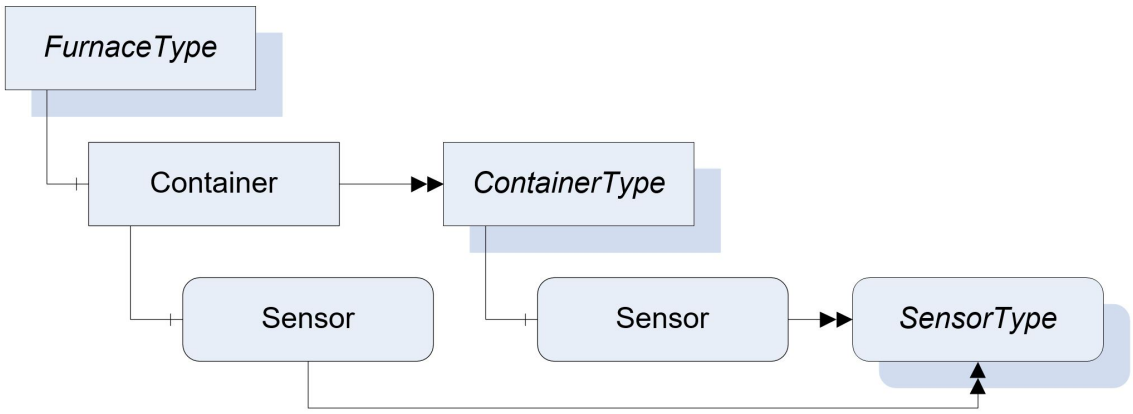

Figure 9: Types describing the sensor example.

4.2.2 Configuration algorithm

In general, determining the TypeDefinition Node whose simulation signal to use for simulating the instance counterpart of a TypeDefinition includes checking one or multiple Nodes. Checking a Node means determining whether the Node has a simulation signal. If a Node with a signal is found, its simulation is used and the rest of the Nodes will be disregarded. On the other hand, if a Node with a configured signal is not found, the instances cannot be simulated based on type configurations. The algorithm for determining the Nodes to be checked and the order in which they are checked proceeds as follows:

1. The TypeDefinition counterpart of the instance is checked. If it is a VariableType, the algorithm ends. If it is a Variable InstanceDeclaration, the next step is performed.

2. The BrowsePath from the type Node having the InstanceDeclaration to the InstanceDeclaration is determined. The BrowsePath consists of the sequence

of Nodes that connect the type and the InstanceDeclaration with hierarchical References. It is possible that there are multiple such paths. Then, the order in which the paths are handled is random, because there is no means for prioritizing the paths.

3. If the BrowsePath includes other InstanceDeclarations between the type Node and the original InstanceDeclaration, they are determined. The order in which the InstanceDeclarations are handled in the next step is the order in which they are mentioned in the BrowsePath.

4. When an InstanceDeclaration is handled, its type is determined and the BrowsePath left from the InstanceDeclaration to the original InstanceDeclaration is determined. Then, it is determined whether an InstanceDeclaration with the aforementioned BrowsePath exists in the type. If it does, the Node is checked. If it does not exist, it might be due to the fact that it is defined in one of the supertypes of the type. Therefore, an InstanceDeclaration with the same BrowsePath is looked for in all the possible supertypes starting from the one at the lowest level in the type hierarchy until such InstanceDeclaration is found or the base type is reached. This way, the InstanceDeclaration that is valid for the fully-inherited InstanceDeclarationHierarchy of the type, i.e., the one at the lowest level in the type hierarchy, is found. However, an InstanceDeclaration with such BrowsePath does not necessarily exist even in any of the supertypes, because the InstanceDeclaration being handled might, for example, be of a simple type even though it has child Nodes.

5. The actual type (a VariableType, the target of a HasTypeDefinition Reference) of the original InstanceDeclaration is checked.

If FurnaceType presented in Figure 9 is instantiated and the counterpart of its Sensor InstanceDeclaration is to be simulated using type configurations, the application of the algorithm for this InstanceDeclaration proceeds as follows: the aforementioned InstanceDeclaration is checked first. It shall be assumed that it does not have a simulation signal. The BrowsePath from FurnaceType to Sensor is /Container/Sensor. Thus, an InstanceDeclaration with the BrowsePath /Sensor is looked for in the type of Container, ContainerType. The Sensor InstanceDeclaration of ContainerType has the aforementioned BrowsePath, so it is checked. It is assumed that it does not have a simulation signal either. Lastly, the type of the first-mentioned Sensor, SensorType, is checked.

4.3 Type instantiation

The SDK uses the type instantiation algorithm designed in an earlier Master's thesis [36]. The algorithm forms the fully-inherited InstanceDeclarationHierarchy of the type being instantiated at run time by browsing Nodes in the Address Space. Then, it creates counterpart Nodes for all of the Nodes defined in the fully-inherited InstanceDeclarationHierarchy of the type. The algorithm eliminates the need to

manually create a counterpart Node for every InstanceDeclaration that a complex type encompasses.

In the thesis, Java source code generation from Information Models defined in UANodeSet XML documents was also introduced. After applying code generation, instances of complex VariableTypes and ObjectTypes can easily be created by server developers using the generated classes that include the information about the structure of the type in question. The instance counterparts of InstanceDeclarations of a complex type can then be accessed from the object presenting an instance of the type using methods that are specifically defined based on the type information.

In this thesis, it is assumed that Information Models are imported to the Simulation Server and corresponding instances created at run time. As generating code from Information Models needs to be performed before compilation, code generation cannot be used in this thesis. However, as the instantiation algorithm is separated from the code generation, it will create all the instances specified by the fully-inherited InstanceDeclarationHierarchy of a type using certain default classes. This results in the child Nodes of an instance corresponding to a complex type not being able to be accessed conveniently, because the type information is not available in the class. Still, this is not a hindrance, because the type information in the class would not even be of any use at run time. All in all, forming fully-inherited InstanceDeclarationHierarchies and creating instances are features that are taken for granted in this thesis.

4.4 Configuring instances

After an instance is created from a type, all of its possible Variable InstanceDeclarations are linked to the counterpart Variable instances. The same applies to the VariableType itself if the instantiated type is a VariableType rather than an ObjectType. It is noteworthy that the InstanceDeclarations may also be defined in a supertype of the type if its fully-inherited InstanceDeclarationHierarchy specifies so. The purpose of the linking is that the values of a simulation signal that a TypeDefinition uses can be set to all the corresponding instances. The instance counterpart of each TypeDefinition is found using the same functionality that is implemented by the TranslateBrowsePathsToNodeIds service. In this case, it always returns only one NodeId, because a created instance contains only the Nodes that are defined by a simple or a complex type.

As mentioned previously, the instance counterparts of a TypeDefinition might be simulated using a simulation signal configured to the TypeDefinition or to another TypeDefinition if the first TypeDefinition is an InstanceDeclaration. This is determined by applying the algorithm presented in section 4.2.2 to the TypeDefinition. After each TypeDefinition is linked to the TypeDefinition whose signal it uses, the values of their instance counterparts can be simulated. In other words, when all the TypeDefinitions that use the signal of a certain TypeDefinition and their instance counterparts are determined, the values provided by the signal are set to all the instances.

4.5 Instance-specific simulation

An instance can be configured with a simulation signal that is specific to the instance. It overrides any simulation that its TypeDefinition counterpart uses. This feature is intended to support the idea that an instance might have different functionality than its equivalents even though being of the same TypeDefinition. If the simulation signal of the TypeDefinition is overridden, this piece of information is added to the linking between the instance and its TypeDefinition, and the values provided by the new signal are set to the instance. The instance-specific simulation signal can be removed from a Node. This will lead to the linking between the instance and its TypeDefinition counterpart to be rendered active again. Thus, the instance will once again be simulated according to the signal that the TypeDefinition counterpart uses.

4.6 Simulation Configuration Information Model

Information Models can be used to expose any kind of data in an Address Space. This thesis introduces an Information Model called Simulation Configuration. It defines types that are built-in to the Simulation Server. Exposing data in the Address Space using the types provided by this Information Model serves two purposes. First, it allows also client applications to infer which simulation signal is configured to a Node and which TypeDefinition is the counterpart of an instance. Second, it allows simulation signals to be reconfigured to Nodes and the linkings between instances and types that provide values for the instances to be re-established when restarting the server. Although all the necessary information regarding simulation is stored to memory by the application, the information is lost when the server is closed. However, when the types of the Simulation Configuration Information Model are instantiated and the instances are saved in the UANodeSet XML format along with the rest of the user-defined Nodes when closing the server, simulation can be resumed based on the information provided by the instances after restart. The types of the Simulation Configuration Information Model are introduced in the following subsections.

4.6.1 ObjectTypes

The different types of simulation signals are described using Objects of complex ObjectTypes containing information about their respective parameters in the values of InstanceDeclaration Variables. Common to all the InstanceDeclarations is that they are Properties, that is, instances of the VariableType PropertyType. The ObjectTypes and their Properties are shown in Figure 10. The abstract Simulation-ConfigurationType, a subtype of BaseObjectType, has all the other ObjectTypes that are used for different simulation signals as its subtypes. An instance of a certain ObjectType is always created when a simulation signal is configured to a Node, and the Object is attached to the Node using a Reference whose type will be presented in the next section. If the user changes the parameters of the signal, the new parameters will be written to the values of the Properties of the Object. When the server is restarted, a simulation signal is created according to the Object and the values of its Properties, and the signal is configured to the Node that refers to the Object. In the

following, the vital Attributes of the Properties of each ObjectType are discussed and presented in tables. These Attributes are the ones that define the value Attribute that contains the value of a simulation signal parameter.

CounterConfigurationType describes the parameters of counter signals, and the Attributes of its Properties are presented in Table 2. The value and the parameters of a counter signal can be presented using objects of multiple Java classes that are subclasses of the abstract class java.lang.Number. These classes are the ones that are used to store values of the built-in, non-abstract subtypes of the DataType Number in the SDK. Therefore, a counter signal can be used to compute values for Nodes of any of the aforementioned DataTypes specifically. In order to accommodate any type of parameter values to the value of the Properties Increment, Initial Value, Maximum Value, and Minimum Value, the DataType of these Properties is Number. The specific DataType of these Properties is defined in the DataType Property as the NodeId of the DataType.

Sawtooth, sinusoid, square, and triangle signals share the common parameters; therefore, the parameters are presented as the Properties of the abstract ObjectType WaveConfigurationType, the Properties of which are shown in Table 3. Having such an abstract type with all the Properties allows not being compelled to define identical Properties in each different subtype that depicts a specific type of a wave signal. The DataType of all the Properties is Double, because regardless of the DataType of the Node that a wave signal is configured to, the parameters and thus also the value of the signal are stored as objects of the class java.lang.Double.

The parameter Properties of ExpressionConfigurationType are presented in Table 4. Expression Property defines the custom formula used by an expression signal, and Links defines the symbols used in the formula and the corresponding NodeIds of Nodes whose values are used in the formula. The length of the two-dimensional array Links is unlimited for the first dimension, that is, there may be any number of symbol-NodeId pairs. For the second dimension, the length is two, because a symbol-NodeId pair consists of two parts. The last signal type, random signal, has no parameters, so RandomConfigurationType has no Properties.

Table 2: Properties of CounterConfigurationType.

BrowseName	**DataType**	**ValueRank**	**ArrayDimensions**
Bidirectional	Boolean	Scalar	-
DataType	NodeId	Scalar	-
Direction Up	Boolean	Scalar	-
Increment	Number	Scalar	-
Initial Value	Number	Scalar	-
Maximum Value	Number	Scalar	-
Minimum Value	Number	Scalar	-

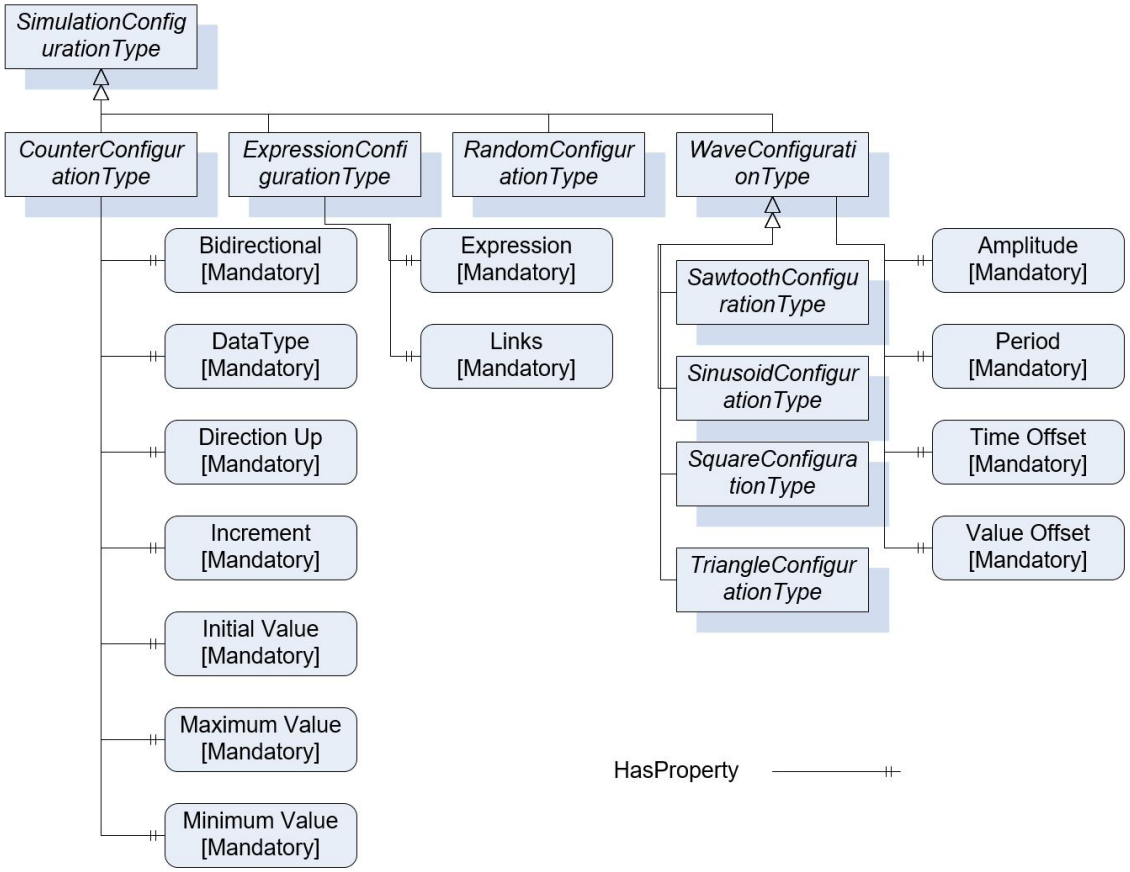

Figure 10: ObjectTypes of the Simulation Configuration Information Model.

Table 3: Properties of WaveConfigurationType.

BrowseName	**DataType**	**ValueRank**	**ArrayDimensions**
Amplitude	Double	Scalar	-
Period	Double	Scalar	-
Time Offset	Double	Scalar	-
Value Offset	Double	Scalar	-

4.6.2 ReferenceTypes

The Simulation Configuration Information Model includes two ReferenceTypes with distinct purposes. The first one is a non-hierarchical type HasSimulationConfiguration, which is a subtype of the abstract type NonHierarchicalReferences. It depicts the relationship between a Node (source) and its Simulation Configuration Object (target). Although it would make sense for it to be a hierarchical ReferenceType, it is determined to be non-hierarchical, because Properties cannot be the source Node of a hierarchical Reference. As HasSimulationConfiguration is non-hierarchical, a Simulation Configuration Object can be configured also to a Property Node.

Table 4: Properties of ExpressionConfigurationType.

BrowseName	**DataType**	**ValueRank**	**ArrayDimensions**
Expression	String	Scalar	-
Links	String	2 Dimensions	[0, 2]

A Simulation Configuration Object referenced by a TypeDefinition Node with a HasSimulationConfiguration Reference is not an InstanceDeclaration, so none of the instance counterparts of the TypeDefinition contain a counterpart for the Object.

The second introduced ReferenceType is called HasTypeSimulation. It is also a non-hierarchical ReferenceType and a subtype of NonHierarchicalReferences. Its semantics is to define the TypeDefinition counterpart (target), according to which a Variable instance (source) is simulated. It is thus used only between an instance and its TypeDefinition if the latter is a VariableType or a Variable InstanceDeclaration. In contrast, HasTypeDefinition Reference refers to the actual type of the instance, and it is not adequate for this purpose, because the desired TypeDefinition may be a Variable InstanceDeclaration that refers to the same type using a HasTypeDefinition Reference. The TypeDefinition counterpart of an instance could actually be determined without a HasTypeSimulation Reference by inspecting the BrowseNames of instances and by searching for Nodes with equal BrowseNames in the types, from which the instances were created. However, in addition to eliminating the necessity to browse several Nodes, having a Reference for this purpose makes it easier for humans to conceive which is the TypeDefinition counterpart of an instance.

5 Creating and simulating data in the Simulation Server

This chapter introduces the new features of the Simulation Server that use Information Models and the concepts that were presented in chapter 4. If the user of the Simulation Server does not have a server whose Information Models they could serialize into an XML document or an Information Model already defined in the UANodeSet format, they can write their own UANodeSet document. One option to perform this is to write Nodes in the UANodeSet format by hand. A more feasible alternative for this is using a tool called UaModeler [37]. It has a graphical user interface that can be used in designing an Information Model by defining Nodes effortlessly. The designed Information Model can be saved to a UANodeSet file.

5.1 Importing Information Models

Information Models defined in the UANodeSet format can be imported to the Simulation Server. The References defined in a document must refer to Nodes that are defined in the same document or already exist in the server. Each Information Model has a set of Nodes that belong to a namespace that is defined by a unique namespace URI. Unique namespace URI ensures that NodeIds and BrowseNames can be differentiated from each other even if several Information Models are used simultaneously. Although the namespace used in the of NodeIds of an Information Model is always the one defined by the Information Model, the BrowseNames may have the namespace of another Information Model. This is the case when the Information Model extends another one, that is, defines subtypes of or uses instances of types defined in another Information Model. When the Nodes belonging to an Information Model are imported, the corresponding namespace URI is added to the NamespaceArray Variable of the server. The respective namespace index is then used in the corresponding NodeIds and BrowseNames. Namespaces that include imported Information Models are from now on called type namespaces, as they primarily contain types.

5.2 Namespaces of the Simulation Server

After an Information Model is imported to the Simulation Server, new instances can be created. The instances can be divided to two categories. First, there are instances that are instantiated according to the types of the imported Information Model. Second, instances of the ObjectTypes and ReferenceTypes defined in the built-in Simulation Configuration Information Model can be created by configuring simulation signals, as explained in section 4.6. These instances are created to different namespaces than the type namespaces containing imported Information Models or the Simulation Configuration Information Model. These namespaces and the motivation for having them will be introduced in the following subsections.

5.2.1 Instance namespaces

Creating instances to specific instance namespaces serves the purpose of separating the instances from the used Information Model that the instances do not actually belong to. For example, if multiple servers that contain instances created from types of the same Information Model are aggregated, the data provided by different servers can be differentiated from each other, because they do not belong to the same namespace. An Information Model can also contain standard instances, but they are not simulated in the Simulation Server, because they are not intended to provide real data. The user can create an instance namespace by defining the URI of the namespace. Then, the namespace URI will be added to the NamespaceArray.

Another way of creating an instance namespace is to import a UANodeSet document that contains instances. This is performed using a different feature compared to when a file containing an Information Model is imported because of the different requirements for processing types and instances. Namely, loading a file that contains instances causes the namespace to be added to the list of namespaces to which instances can be created, whereas loading a file containing an Information Model does not. Also, loading an Information Model leads to the simulation configuration algorithm to be applied to the TypeDefinitions of the Information Model. The reason for this is that even though simulation signals are not yet configured to these TypeDefinitions, they might use a simulation signal configured to a TypeDefinition defined in another Information Model.

5.2.2 Simulation configuration namespaces

For each namespace that includes an imported Information Model, another namespace is created. This is the namespace that contains the Simulation Configuration Objects of the TypeDefinitions of the Information Model. The index of a simulation configuration namespace is always one greater than the one of the Information Model. The reason for separating the TypeDefinitions from their Simulation Configuration Objects is the same as for separating instances from their types, that is, the Simulation Configuration Objects are not part of the Information Model whose Nodes refer to the Objects. Also, as they belong to different namespaces, it becomes convenient to differentiate the Nodes from each other when they are to be serialized to separate files.

A namespace for simulation configurations is created also for each instance namespace. When the type-based simulation is overridden in an instance by configuring an instance-specific simulation signal, a corresponding Simulation Configuration Object is created to the aforementioned namespace. Even though the simulation signal used by the TypeDefinition of the instance no longer provides new values for the instance, the HasTypeSimulation Reference to the TypeDefinition is not removed. This enables inferring (e.g., when restarting the server) merely by inspecting Nodes and References that even though a specific simulation is configured to the instance, it can still later be simulated with respect to its TypeDefinition.

5.3 Simulation View

The features related to simulating instances created from Information Models are located in the Simulation View of the user interface of the Simulation Server. Compared to the first version of the Simulation Server, the most prominent modification is that there is a view presenting the Nodes of the Address Space as a tree-like structure instead of a list of simulated Variables that are located in a single Folder. Having such a view makes configuring simulation to Nodes and monitoring values of multiple Nodes simultaneously effortless. The icons in the tree show the NodeClass of a Node, and the name shown is DisplayName, which is an Attribute that defines the localized name of a Node.

The simulation-related features presented in chapter 4 are used by means of manipulating Nodes using the different columns of a table view that the tree view is a part of. The functionalities attached to the columns will be presented in the following subsections. All the other features of the Simulation View, i.e., changing parameters of simulation signals, plotting values of Variables, showing simulation time, changing simulation interval, and starting and pausing simulation have not been changed from the original version. Using the features presented in this section, however, requires simulation to be paused. Also, the figures of the Simulation View that will be presented do not illustrate the appearance of the final version of the server, because the version designed in this thesis is a prototype, and the internal functionality is emphasized in this scope rather than the user interface features.

5.3.1 Configuring types

The most important column of the table view is the Signal Type column. For each row, it contains a combo box that includes the types of simulation signals that can be configured to a Node. When a simulation signal is configured to or removed from a VariableType or a Variable InstanceDeclaration, the algorithm presented in section 4.2.2 is applied to Nodes so that the linkings between TypeDefinitions that use the signals of each other are re-established. These changes will reflect to all the instances regardless of whether they are already created or will be created later.

Another important column is the DataType column, which shows the DataType of Variables and VariableTypes. When it comes to TypeDefinitions, DataType cannot be altered from the column. On the contrary, it is assumed that the imported Information Models are complete so that the DataType and other Attributes of TypeDefinitions do not need to be modified after importing them. Simulation can be configured to TypeDefinitions whose DataType is one of the built-in DataTypes that inherit from Number (including Number). If the DataType is not a built-in, non-abstract subtype of Number, a counter signal may not be configured to the Node, because counter signals are specific to the aforementioned DataTypes. An example ObjectType with simulation signals configured to its InstanceDeclaration Variables is shown in Figure 11.

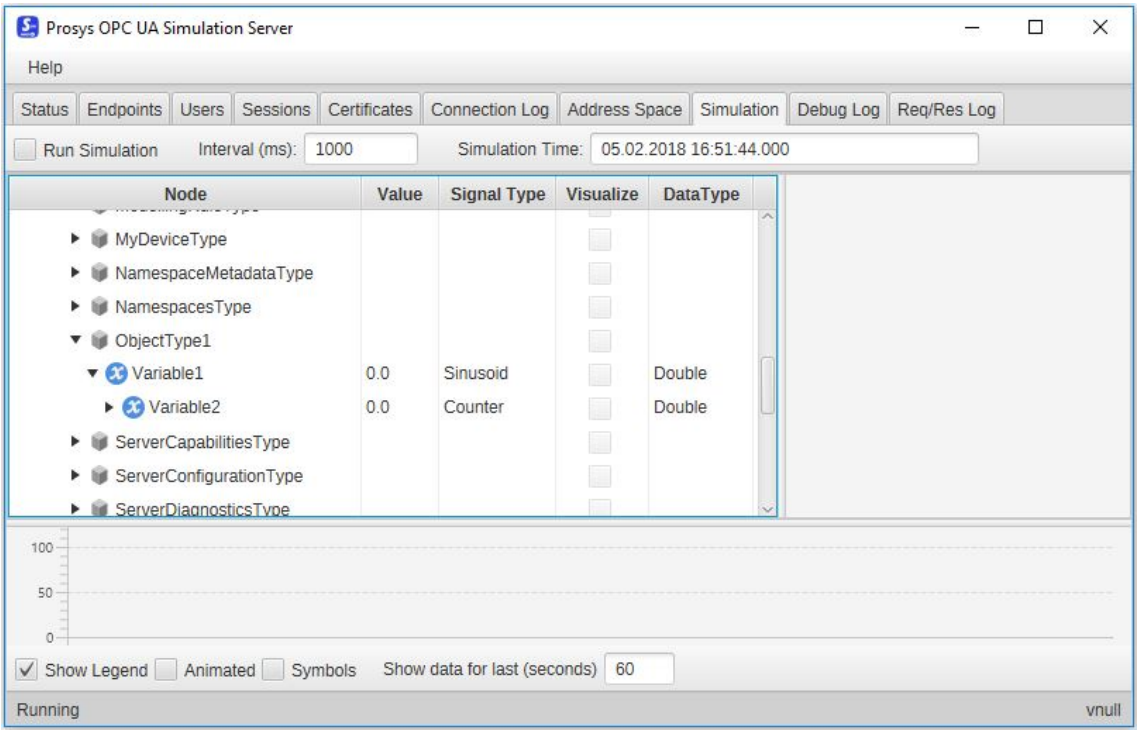

Figure 11: Simulation signals configured to Variable InstanceDeclarations.

5.3.2 Creating instances

The user can create instances of VariableTypes and ObjectTypes at run time according to the TypeDefinitions of the imported Information Models in a dialog that is opened from the context menu of a type Node in the tree view. The dialog is shown in Figure 12. The type that is being instantiated is ObjectType1, which is visualized using the graphical notation in Figure 13. An XML document that could be used to describe the Nodes presented in Figure 13 is provided as an example in Appendix C. The instantiation dialog contains several options. The user must define the base name for the instance, the amount of instances to be created, and the instance namespace to which the instance is created. The base name will be assigned as the BrowseName, the DisplayName, and the identifier part of the NodeId of the instance. If the NodeId using the base name exists already, the base name is extended by the smallest number for which the NodeId is vacant.

The dialog also includes a list of the InstanceDeclarations with Optional ModellingRule, which stems from the generated fully-inherited InstanceDeclarationHierarchy. The user may select which ones of these InstanceDeclarations will have counterparts in the instance. If an Optional InstanceDeclaration is selected, one must also select all the other Optional InstanceDeclarations that are used to connect the first InstanceDeclaration to the type Node. If an Optional InstanceDeclaration is not chosen, any Node beneath it will not have a counterpart in the instance. The purpose of the checkbox below the namespace selection spinner is to specify whether the Variables of the instance will be simulated according to their TypeDefinition

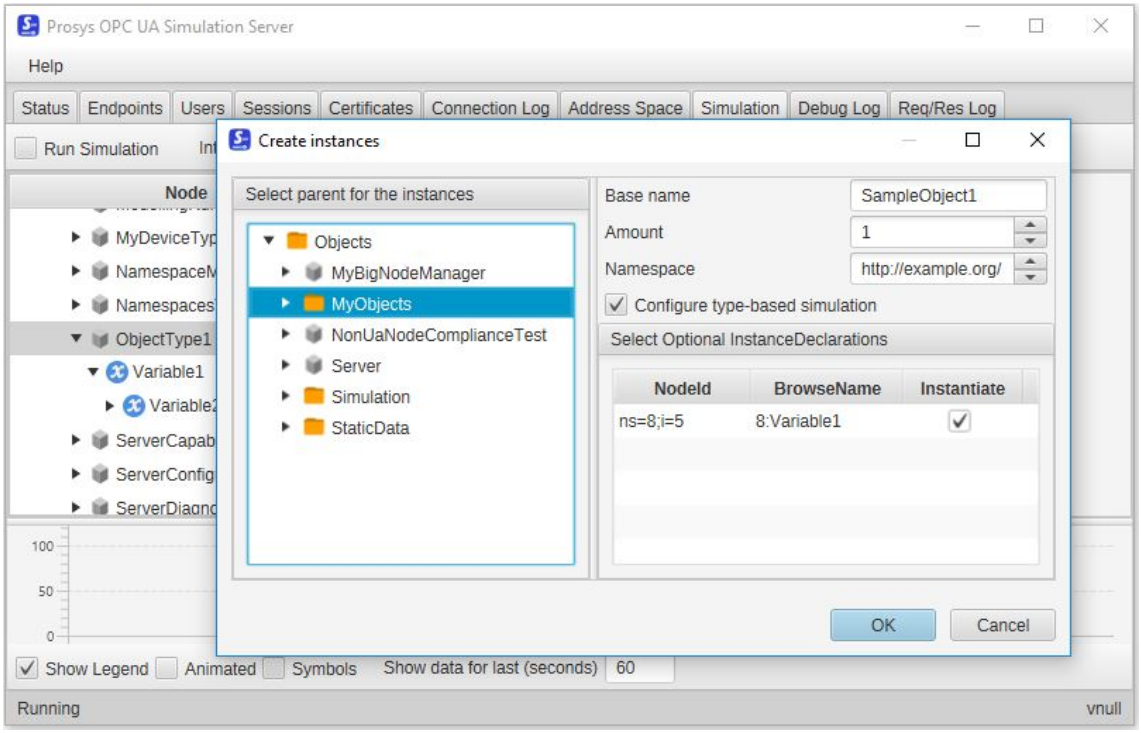

Figure 12: Creating an instance.

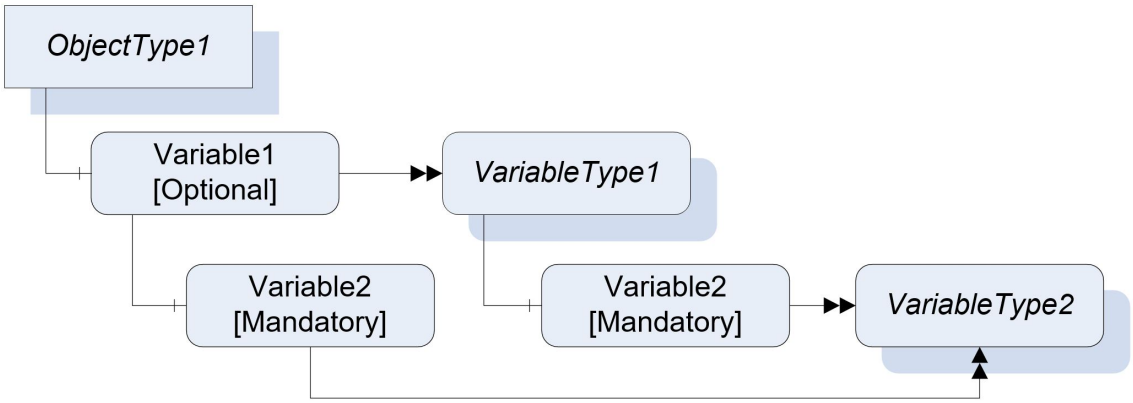

Figure 13: Types used in the instantiation example.

and connected to the TypeDefinition using a HasTypeSimulation Reference. The last parameter to be determined is the Object or Variable that will be the parent Node of the instance. This is conducted by selecting the Node in the pane on the left. Mahnke, Leitner, and Damm [2] have listed some of the best practices that can be applied in planning the structure of the Objects Folder.

5.3.3 Simulating instances

The values of simulated Variable instances can be monitored in the Value column and plotted using the Visualize column, as shown in Figure 14. Currently, the Simulation

Server supports simulation of scalar values only. Because simulation signals in type-based simulation are specific to TypeDefinitions, the simulation parameters can only be altered in the TypeDefinition that a signal is configured to. Overriding the type-based simulation in an instance is performed simply by selecting a simulation signal for the instance. Then, the parameters of the new signal can be changed any time. Reverting to the type-based simulation is conducted by selecting an option for this operation in the context menu.

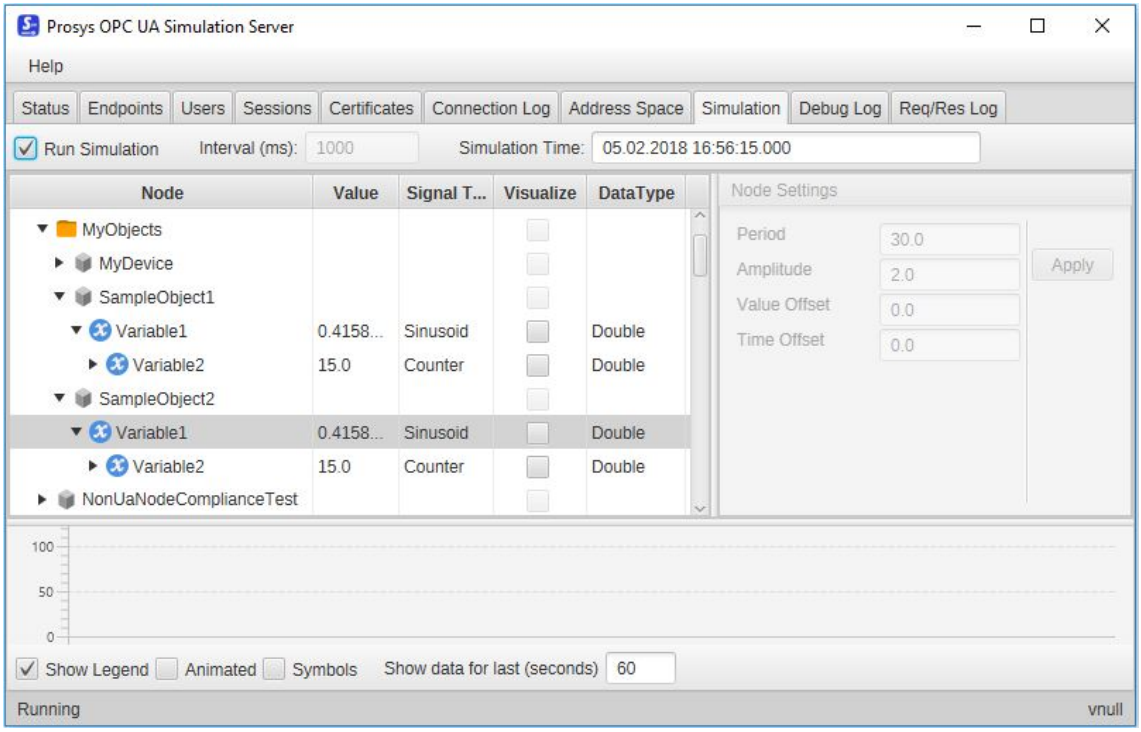

Figure 14: Simulating instances using type configurations.

As explained in section 4.6.1, different types of simulation signals store their values using objects of different Java classes. All signals except counters use the class java.lang.Double, which is why the values need to be converted if the DataType of the instance to which the value is set is not written as a Double. The conversion is performed using a data type converter, which is a feature of the SDK. It takes the DataType of the instance and the value object as parameters, and returns a converted value. Certain DataTypes are mapped to the respective classes that they use, which is how the converter knows which class the value should be converted to.

If an instance has an abstract DataType, the conversion becomes troublesome. Because abstract DataTypes do not have a class mapped to them, the value cannot be converted. Further, the non-converted value might not be suitable for the instance, because a value of no DataType can be set to a Node of an abstract DataType if the first-mentioned DataType is not a subtype of the latter. For example, a Double value cannot be set to a Node whose DataType is Integer. For the aforementioned reasons, the DataType of an instance can be changed in the DataType column into a built-in, non-abstract subtype of the DataType of its TypeDefinition (only if the

latter type inherits from Number) in order to support converting values to the correct type. This also results in the benefit that a counter signal can be configured to a wider range of Nodes, because one cannot be configured to a Node of an abstract DataType.

When simulating an instance, the simulated value must be in the range of values that its DataType is able to describe. Otherwise, the value will not be set to the instance. Because a counter signal is specific to a DataType, its parameters are automatically validated when they are changed. If any of the parameters is not in the range of the DataType, the changes are reverted, which is why counters will not create values that are out of the range of the DataType. One may also define additional constraints to the range of possible values in addition to those set by the DataType of the instance. For example, the VariableType AnalogItemType, which is part of the base Information Model, involves such constraints.

5.4 Server configuration

The configuration of the Simulation Server is defined in an XML document settings.xml, which is automatically created when the server is started for the first time [1]. The settings consist of the configurable parameters of the server, which include, for example, the supported transport protocols, the network ports that the protocols use, and the supported security models and security policies. When the server is started, the content of the document is unmarshalled into a singleton object using JAXB, and the server is configured according to the unmarshalled data. While the server is running, the variables of the object are modified according to the configuration-altering actions performed by the user. The content of the object is marshalled to the same XML document when the server is closed so that the same settings are preserved when the server is restarted.

5.4.1 Saving and loading Nodes

The server configuration file can also be used to save imported Information Models, created instances, and simulation signal configurations when the server is closed so that they can be recreated automatically during restart. Saving Nodes is performed with the serialization feature designed in chapter 3. The information about the files to which Nodes are saved is integrated to the configuration file. Namely, always when a new namespace is created to the server, a new entry is added to the singleton settings object. Such entry contains the URI of the namespace, information about whether the namespace is an instance namespace, and the name of an XML file. The last-mentioned defines the file that the namespace will be saved to (saving a namespace is used to refer to saving the Nodes of the namespace). This file is located in the same folder as the settings file. Thus, Nodes of all the namespaces that are not built-in to the server are saved to individual files when the server is closed. When the server is restarted, the files are loaded and processed as required by the type of their content. The order in which Nodes of different namespaces will be saved and loaded is based on the order that the namespaces were created.

Because Nodes are saved to different files according to their namespace, it is vital to consider the dependencies between these documents. Namely, the References between Nodes of different namespaces need to be written to the Node that requires the other Node to exist and not vice versa. As explained in section 3.7.1, this generally requires that hierarchical References are written to the target Node and non-hierarchical References to the source Node. A document containing an Information Model may refer to another Information Model that it extends with non-hierarchical HasTypeDefinition References and be referred to by it with hierarchical HasSubtype References. All these References are written to the first-mentioned document that will be read after the other one. Therefore, no dependency issues will be caused. This is also how Information Models are defined in the first place.

The relationships between instances and types are established with non-hierarchical References, such as HasTypeDefinition and HasTypeSimulation. As the source Node is an instance and the target Node a TypeDefinition in such References, there will be no issues in reading a file that contains instances if the files containing the required types are read first. However, an instance namespace may have a smaller index than the types that the instances refer to. Thus, all the files containing types are read before any of the files containing instances. Nevertheless, the indexes of the namespace URIs in the NamespaceArray are preserved.

When it comes to simulation configurations, the non-hierarchical HasSimulation-Configuration References are written by default to the source Node that owns a Simulation Configuration Object (the target Node). The namespace of the source is always one smaller than the one of the target, and the file that contains the source is thus read before the one that contains the target. This will cause a dependency conflict, because the file that is read first contains a Reference to an Object that is unknown when the file is read. Therefore, HasSimulationConfiguration References are exceptionally written to the target Node.

5.4.2 Multiple configurations

The configuration file is located in a fixed default location, which is known by the Simulation Server so that it can read from and write to the file in that location. As such, there cannot be multiple different sets of settings to be used, unless a path to another settings file is provided. Therefore, the user can specify another folder that contains a configuration file to be used as a command line parameter. The file from which the server configuration is read during start-up is always the same that the configuration is written to when the server is closed. Hereby, multiple configurations can be defined in different locations. When the server is started using configuration in a certain location, the configuration will also have its own certificate folders and a database to store a log of connections that are established to the server using the configuration. The possibility to have multiple configurations also allows running multiple instances of the Simulation Server simultaneously on the same machine. However, different network ports need to be selected for each configuration that is used to run an instance of the server so that the instances can coexist.

The main benefit of supporting multiple configurations, however, is that it enables

several different sets of Nodes to be automatically loaded to the server when it is started. The user can define different combinations of Information Models, instances, and simulation configurations by running the server using different configurations and creating the Nodes that will then be specific to the configuration in question. Additionally, because all the files related to a configuration are located in the same folder, the configuration can conveniently be copied to another location. Then, the complete configuration can be used for running the Simulation Server also from that location.

6 Conclusions and future work

This thesis examined the serialization of the Address Space of an arbitrary OPC UA server from data provided as Java objects into a machine-readable XML format and configuring simulation signals to Nodes deserialized from the XML data in the Simulation Server. A solution to the first research question of how to serialize the data of an Address Space was proposed. The solution includes browsing the Nodes of the Address Space, reading the data related to the Nodes, creating Java objects, and marshalling the objects into XML data using JAXB. The objects were created from JAXB classes that were automatically generated from XML schemas, which eliminates the risk of creating invalid XML data. The used schemas are of the standardized UANodeSet format, so the data created according to them is general. Currently, using the serialization requires the user to select the namespaces whose Nodes are saved to a single file. This could be improved by allowing the user to choose whether different namespaces shall be saved to different files. Then, the user would not be required to save all Nodes that are written to different files separately.

Another feature that would greatly advance the serialization feature would be to automatically save the Nodes of all the other namespaces to which the Nodes of a selected namespace have References. This requires forming namespace hierarchies of the server, that is, determining all the namespaces that the Nodes of certain namespaces have References to directly or via other Nodes. This will result in information about which namespaces depend on other ones. Thus, the responsibility for serializing all the Nodes that are required in order to deserialize certain other serialized Nodes is shifted from the user to the application.

The original Simulation Server introduced the concept of creating simple Variables whose value can be simulated using different types of simulation signals. The answer to the second research question of how custom Information Models are to be combined with simulation lies in developing the simulation feature of the original Simulation Server. The feature was significantly improved by introducing the possibility to diversely configure the same simulation signals to TypeDefinitions that are defined in imported Information Models. The new features also include instantiating the imported types and simulating the instances according to signals configured to the TypeDefinitions or to signals that are configured specifically to individual instances. The presented solution covered all the essential aspects of the aforementioned functionalities. Although the new features allow having more meaningful data in the Simulation Server, there are several aspects from which incrementing the number of benefits achieved using the application can be approached.

The principal downside of the Simulation Server is that the simulated signals are extremely rudimentary and useful only for processes that do not require an accurate simulation model. This issue can be diminished in a number of ways. The first and foremost measure to be taken in order improve the usability of the Simulation Server is to design more accurate simulation models. These models could involve, for example, simulating values that are not numbers, such as booleans or strings. It could also be allowed for users to define their own simulation models, for example, by integrating them to the imported Information Models in the UANodeSet format.

The simulation features do not have to be limited to simulating values of Nodes. The OPC UA specification introduces, among other things, events, alarms, and conditions, the simulation of which could also be beneficial. Another useful feature would be the possibility to repeat the value history of a Variable. The provided history could be either history of a Node that is already simulated in the Simulation Server or history provided by another server that could be imported to the Simulation Server from a database. Compared to reading the value history of a Node, this feature would establish the advantage that changes in the value Attribute can be monitored and visualized as if they occurred real-time.

There are also a number other of improvements to be applied that are related to better utilization of OPC UA. First, simulation could be changed so that it will be based on Subscriptions. This means that a simulated value is set to an instance only if a client application subscribes to the changes of its value using the Subscription service set in order prevent setting values to no avail. Also, different tools to assist building a more customized Address Space shall be added. These include, for example, the ability to create Folders for organizing the Address Space, to connect Nodes with References, and to specify Attributes for instances during instantiation.

This thesis also introduced the feature of saving the imported Information Models, created instances, and simulation signal configurations automatically to multiple files when the server is closed using the designed serialization feature. The information about the files was integrated to the server configuration file that includes all the server parameters. This is the answer to the third research question, which was about how user-defined Nodes can automatically be restored after the server is closed and then restarted. By supporting multiple different server configurations that are defined in different folders, the user can flexibly start the Simulation Server using different configurations. The configuration used to run the server is specified using command line parameters, which is somewhat clumsy. A user-friendly way for achieving the same outcome could be having a start view in which a configuration is selected, after which the server is actually started. In the same view, there could be options to create and modify configurations so that the user will not be compelled to start the server using a configuration before modifying it. This feature would be worthwhile also because it could possibly be used in other server applications.

The last aspect to be considered in improving the application is its usability. As the software designed in this thesis is merely a prototype in which the internal features have been emphasized, certain usability considerations are still required to be dealt with. The approach to this will be to make the graphical user interface as easy and intuitive to use as possible. All in all, this thesis built the foundations for the utilization of the concept that data created from a specific Information Model can be simulated in a test environment according to simulation signals that are configured to types or instances. Designing any of the aforementioned prospective improvements will develop the Simulation Server from a proof-of-concept application towards a test tool that will be useful in several domains.

References

[1] B. Boström, "JavaFX based OPC UA Simulation Server," Master's thesis, School of Electrical Engineering, Aalto University, Espoo, 2014.

[2] W. Mahnke, S.-H. Leitner, and M. Damm. *OPC Unified Architecture*. Springer, 2009.

[3] R. Santos, J. Normey-Rico, A. Gómez, L. Arconada, and C. de Prada Moraga, "OPC based distributed real time simulation of complex continuous processes," *Simulation Modelling Practice and Theory*, vol. 13, no. 7, pp. 525-549, Oct. 2005.

[4] M. Mahmoud, M. Sabih, and M. Elshafei, "Using OPC technology to support the study of advanced process control," *ISA Transactions*, vol. 55, pp. 155-167, Mar. 2015.

[5] T. Miettinen, "Synchronized Cooperative Simulation: OPC UA Based Approach," Master's thesis, School of Electrical Engineering, Aalto University, Espoo, 2012.

[6] "Classic," OPC Foundation. [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-classic/. [Accessed: Dec. 27, 2017].

[7] "What is OPC?," OPC Foundation. [Online]. Available: https://opcfoundation.org/about/what-is-opc/. [Accessed: Dec. 27, 2017].

[8] "Unified Architecture," OPC Foundation. [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/. [Accessed: Dec. 27, 2017].

[9] V. Van Tan and M.-J. Yi, "OPC UA Based Information Modeling for Distributed Industrial Systems," in *International Conference on Intelligent Computing*, 2010, pp. 531-539.

[10] "OPC UA Java SDK," Prosys OPC Ltd. [Online]. Available: https://www.prosysopc.com/products/opc-ua-java-sdk/. [Accessed: Dec. 28, 2017].

[11] "OPCFoundation/UA-Java: The official OPC Foundation Unified Architecture Java Stack Implementation," GitHub. [Online]. Available: https://github.com/OPCFoundation/UA-Java. [Accessed: Dec. 28, 2017].

[12] "OPC Unified Architecture Specification, Part 4: Services, Release 1.03," OPC Foundation. 2015.

[13] "OPC Unified Architecture Specification, Part 7: Profiles, Release 1.03," OPC Foundation. 2015.

[14] "OPC Unified Architecture Specification, Part 6: Mappings, Release 1.03," OPC Foundation. 2015.

[15] "OPC Unified Architecture Specification, Part 3: Address Space Model, Release 1.03," OPC Foundation. 2015.

[16] "OPC Unified Architecture Specification, Part 5: Information Model, Release 1.03," OPC Foundation. 2015.

[17] "OPC Unified Architecture for Devices (DI), Release 1.01," OPC Foundation. 2013.

[18] "OPC Unified Architecture for Analyzer Devices (ADI), Release 1.01a," OPC Foundation. 2015.

[19] "OPC Unified Architecture / PLCopen Information Model, Release 1.00," OPC Foundation. 2010.

[20] A. Fernbach, W. Granzer, and W. Kastner, "Interoperability at the Management Level of Building Automation Systems: A Case Study for BACnet and OPC UA," in *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, 2011, pp. 1-8.

[21] S. Lehnhoff, S. Rohjans, M. Uslar, and W. Mahnke, "OPC Unified Architecture: A Service-Oriented Architecture for Smart Grids," in *Proceedings of the First International Workshop on Software Engineering Challenges for the Smart Grid*, 2012, pp. 1-7.

[22] O. Palonen, "Object-oriented implementation of OPC UA information models in Java," Master's thesis, School of Science and Technology, Aalto University, Espoo, 2010.

[23] "Opc.Ua.NodeSet2.xml," OPC Foundation. [Online]. Available: https://opcfoundation.org/UA/schemas/1.03/Opc.Ua.NodeSet2.xml. [Accessed: Jan. 10, 2018].

[24] "OPC UA Client," Prosys OPC Ltd. [Online]. Available: https://www.prosysopc.com/products/opc-ua-client/. [Accessed: Dec. 27, 2017].

[25] "Extensible Markup Language (XML)," W3C. [Online]. Available: https://www.w3.org/XML/. [Accessed: Jan. 11, 2018].

[26] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "XML Schema Part 1: Structures Second Edition," W3C, Oct. 28, 2004. [Online]. Available: https://www.w3.org/TR/xmlschema-1/. [Accessed: Jan. 11, 2018].

[27] P. Biron and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition," W3C, Oct. 28, 2004. [Online]. Available: https://www.w3.org/TR/xmlschema-2/. [Accessed: Jan. 11, 2018].

[28] "UANodeSet.xsd," OPC Foundation. [Online]. Available: https://opcfoundation.org/UA/schemas/1.03/UANodeSet.xsd. [Accessed: Jan. 12, 2018].

[29] "Opc.Ua.Types.xsd," OPC Foundation. [Online]. Available: https://opcfoundation.org/UA/schemas/1.03/Opc.Ua.Types.xsd. [Accessed: Jan. 12, 2018].

[30] E. Ort and B. Mehta, "Java Architecture for XML Binding (JAXB)," Oracle, Mar. 2003. [Online]. Available: http://www.oracle.com/technetwork/articles/javase/index-140168.html. [Accessed: Jan. 16, 2018].

[31] "Java Architecture for XML Binding," Oracle. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxb/index.html. [Accessed: Feb. 28, 2018].

[32] "JAXB Architecture," Oracle. [Online]. Available: https://docs.oracle.com/javase/tutorial/jaxb/intro/arch.html. [Accessed: Jan. 16, 2018].

[33] "xjc," Oracle. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/xjc.html. [Accessed: Jan. 17, 2018].

[34] "Binding XML Schemas," Oracle. [Online]. Available: https://docs.oracle.com/javase/tutorial/jaxb/intro/bind.html. [Accessed: Jan. 17, 2018].

[35] "Customizing JAXB Bindings," Oracle. [Online]. Available: https://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.5/tutorial/doc/JAXBUsing4.html. [Accessed: Jan. 17, 2018].

[36] E. Laukkanen, "Java source code generation from OPC UA information models," Master's thesis, School of Electrical Engineering, Aalto University, Espoo, 2013.

[37] "UaModeler "Turns Design into Code"," Unified Automation GmbH. [Online]. Available: https://www.unified-automation.com/products/development-tools/uamodeler.html. [Accessed: Jan. 18, 2018].

A xjc customization example

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxb:bindings jaxb:version="2.0"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <jaxb:globalBindings>
      <xjc:javaType name="org.opcfoundation.ua.builtintypes.
         UnsignedByte" xmlType="xs:unsignedByte"
        adapter="thesis.example.UnsignedByteAdapter" />
    <xjc:javaType name="org.opcfoundation.ua.builtintypes.
       UnsignedInteger" xmlType="xs:unsignedInt"
        adapter="thesis.example.UnsignedIntegerAdapter" />
    <xjc:javaType name="org.opcfoundation.ua.builtintypes.
       UnsignedLong" xmlType="xs:unsignedLong"
        adapter="thesis.example.UnsignedLongAdapter" />
    <xjc:javaType name="org.opcfoundation.ua.builtintypes.
       UnsignedShort" xmlType="xs:unsignedShort"
        adapter="thesis.example.UnsignedShortAdapter" />
  </jaxb:globalBindings>
</jaxb:bindings>
```

B XmlAdapter example

```
package thesis.example;

import javax.xml.bind.annotation.adapters.XmlAdapter;
import org.opcfoundation.ua.builtintypes.UnsignedInteger;

public class UnsignedIntegerAdapter extends XmlAdapter<String,
   UnsignedInteger> {

  @Override
  public String marshal(UnsignedInteger v) {
    return v != null ? v.toString() : null;
  }

  @Override
  public UnsignedInteger unmarshal(String v) {
    return UnsignedInteger.parseUnsignedInteger(v);
  }

}
```

C UANodeSet XML document example

```
<?xml version="1.0" encoding="utf-8"?>
<UANodeSet xmlns="http://opcfoundation.org/UA/2011/03/UANodeSet.xsd
   " xmlns:types="http://opcfoundation.org/UA/2008/02/Types.xsd">
  <NamespaceUris>
    <Uri>http://example.org/types/</Uri>
  </NamespaceUris>
  <Aliases>
    <Alias Alias="Double">i=11</Alias>
    <Alias Alias="HasModellingRule">i=37</Alias>
    <Alias Alias="HasTypeDefinition">i=40</Alias>
    <Alias Alias="HasSubtype">i=45</Alias>
    <Alias Alias="HasComponent">i=47</Alias>
  </Aliases>
  <UAVariableType DataType="Double" NodeId="ns=1;i=1" BrowseName="1
     :VariableType2">
    <DisplayName>VariableType2</DisplayName>
    <References>
      <Reference ReferenceType="HasSubtype" IsForward="false">i=63<
         /Reference>
    </References>
  </UAVariableType>
  <UAVariableType DataType="Double" NodeId="ns=1;i=2" BrowseName="1
     :VariableType1">
    <DisplayName>VariableType1</DisplayName>
    <References>
      <Reference ReferenceType="HasSubtype" IsForward="false">i=63<
         /Reference>
    </References>
  </UAVariableType>
  <UAVariable DataType="Double" NodeId="ns=1;i=3" BrowseName="1
     :Variable2">
    <DisplayName>Variable2</DisplayName>
    <References>
      <Reference ReferenceType="HasModellingRule">i=78</Reference>
      <Reference ReferenceType="HasComponent" IsForward="false">ns
         =1;i=2</Reference>
      <Reference ReferenceType="HasTypeDefinition">ns=1;i=1</
         Reference>
    </References>
    <Value>
      <types:Double>0</types:Double>
    </Value>
  </UAVariable>
  <UAObjectType NodeId="ns=1;i=4" BrowseName="1:ObjectType1">
    <DisplayName>ObjectType1</DisplayName>
    <References>
      <Reference ReferenceType="HasSubtype" IsForward="false">i=58<
         /Reference>
    </References>
  </UAObjectType>
  <UAVariable DataType="Double" NodeId="ns=1;i=5" BrowseName="1
     :Variable1">
```

```
    <DisplayName>Variable1</DisplayName>
    <References>
      <Reference ReferenceType="HasModellingRule">i=80</Reference>
      <Reference ReferenceType="HasComponent" IsForward="false">ns
         =1;i=4</Reference>
      <Reference ReferenceType="HasTypeDefinition">ns=1;i=2</
         Reference>
    </References>
    <Value>
      <types:Double>0</types:Double>
    </Value>
  </UAVariable>
  <UAVariable DataType="Double" NodeId="ns=1;i=6" BrowseName="1
     :Variable2">
    <DisplayName>Variable2</DisplayName>
    <References>
      <Reference ReferenceType="HasModellingRule">i=78</Reference>
      <Reference ReferenceType="HasComponent" IsForward="false">ns
         =1;i=5</Reference>
      <Reference ReferenceType="HasTypeDefinition">ns=1;i=1</
         Reference>
    </References>
    <Value>
        <types:Double>0</types:Double>
    </Value>
  </UAVariable>
</UANodeSet>
```